CoGui User Guide

Table des matières

Basics

This new version 3 migrates CoGui from a classic Java application to a completely different architecture based on the NetBeans Platform
If you discover CoGui, a good way is to follow Getting Started section.
CoGui 2 will always be available for download here: http://www.lirmm.fr/cogui/cogui_2.0b6.jar but it is recommended to install the new version. All the instructions about the installation are available in Installation section.
The application CoGui installed will allow you to automatically update CoGui over the corrections and improvements.
If you have some old CoGui projects from before v3.0 (COGXML format) you can import them to the new project format, read more about projects import in Import, Export and Convert section.

The user guide is divided as follows:

- Knowledge representation
- Reasoning
- Import, Export and Convert
- Building documents
- Extending CoGui

Installation

Pre-Requirements

CoGui 3.1 requires Java 8 or newer installed on your computer (PC, Unix/Linux or MacOS).
All the various distributions can be found in Download CoGui section.

Install CoGui using the windows installer

Once you have downloaded the installer file, double-click the file to start the installation wizard.

Install CoGui on other operating systems

Once you have downloaded the ZIP file, unpack your archive using the utilities appropriate for your system.
To launch CoGui 3.0, navigate to the bin sub-directory of your CoGui installation and execute the launcher that is appropriate for your system (windows file "cogui*.exe" or script "cogui").

Keyboard shortcuts

Finding, Searching, and Replacing

Ctrl-F3	Search word at insert point
F3/Shift-F3	Find next/previous in file
Ctrl-F/H	Find/Replace in file
Ctrl-Shift-F/H	Find/replace in projects
Alt-Shift-H	Turn off search result highlights
Ctrl-R	Rename
Ctrl-U, then U	Convert selection to uppercase
Ctrl-U, then L	Convert selection to lowercase
Ctrl-U, then S	Toggle case of selection
Ctrl-Shift-V	Paste formatted
Ctrl-Shift-D	Show Clipboard History
Ctrl-I	Jump to quick search field
Alt-Shift-L	Copy file path
Ctrl-Enter	Triggers completion tool (Factory view)

Opening and Toggling between Views

Ctrl-Tab (Ctrl-`)	Switch between open documents by order used
Shift-Escape	Maximize window (toggle)
Ctrl-F4/Ctrl-W	Close selected window
Ctrl-Shift-F4	Close all windows
Shift-F10	Open contextual menu
Ctrl-PgUp / PgDown	Switch between open documents by order of tabs
Ctrl-Alt-T	Reopen recently closed file

Editing with graphical editors

Ctrl-C	Copy selected vertices and edges
Ctrl-V	Pasted vertices and edges
Del	Delete selected vertices and edges. It also delete pending edges if any
Ctrl-Mouse	To create edges between concept types or relation types into vocabulary graphical editor
Ctrl-Wheel	Zoom +/-

Knowledge representation

CoGui works on a model of a knowledge base consisting of:

The ontological part
- A unique and necessary Vocabulary
- A set of Individuals
- A set of Rules
- A set of Constraints

Data organized into
- A set of Facts
- A set of Queries

Vocabulary

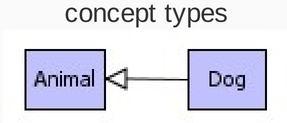
CoGui is able to create multilingual ontologies designed for Conceptual Graphs (CGs). A CG Ontology is composed of exact knowledge and contextual knowledge. The vocabulary is one important part of the exact knowledge and consists of two hierarchies:
1. a hierarchy of concept types (also named concept or class or object type)
2. a hierarchy of relation types (also named relation) with arity greater or equal to 1.

The above hierarchies are respectively organized in partially ordered sets (not necessarily a tree or a lattice). The exact knowledge of the ontology, apart from the vocabulary, consists of:
- a collection of individuals
- rules

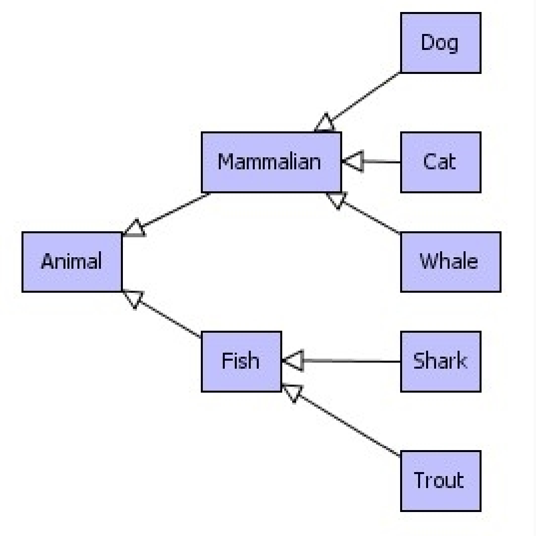
Editors allow end users to navigate through the ontology and edit graphically its structure and content. The ontology is controlled and, if necessary, tools are provided to correct it.

Graphically, types are displayed as vertices. An arc connecting vertex A to vertex B means that the type A is a kind of type B (or A is a specialization of B or B is a generalization of A):

concept types

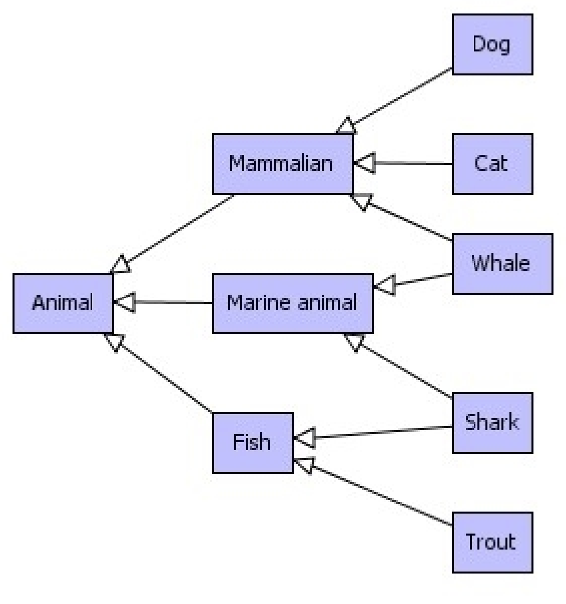


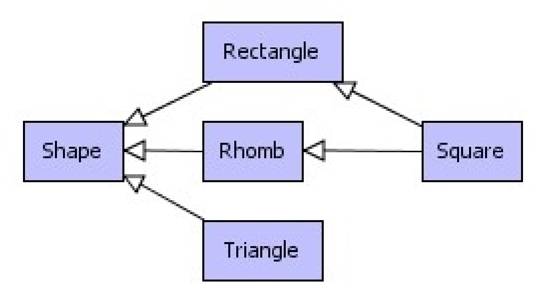
In most cases the ordered set looks like this:



simple ordered set of concept types

In this case, the hierarchical structure is a tree. But the model accepts extra connections. Two examples below illustrate hierarchies that not have a tree structure:

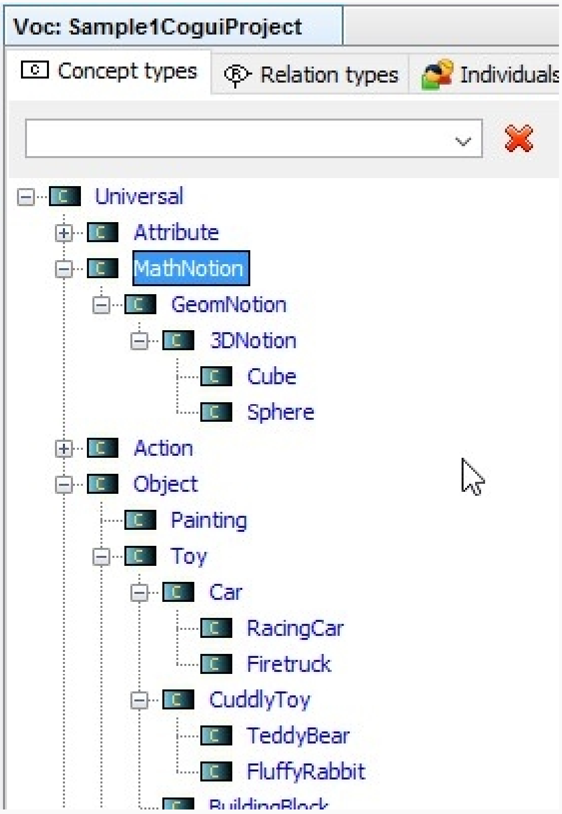




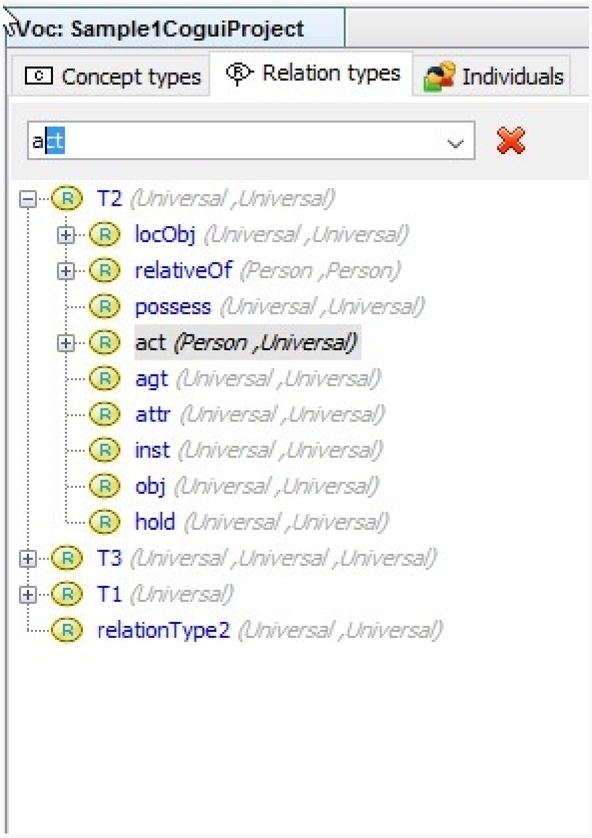
The edit operation is not heavily constrained by the model, in practice, the only critical error occurs when a circuit is detected. More details can be found in following chapters.

How to browse through type hierarchies

When a project is opened (or created) a vocabulary panel appears on the right part of the main window. Concept types, relation types and Individuals are displayed in three separated tabbed panels. An arborescent representation containing every path between maximal type and others. Types are alphabetically sorted, relation types are also sorted by arity.

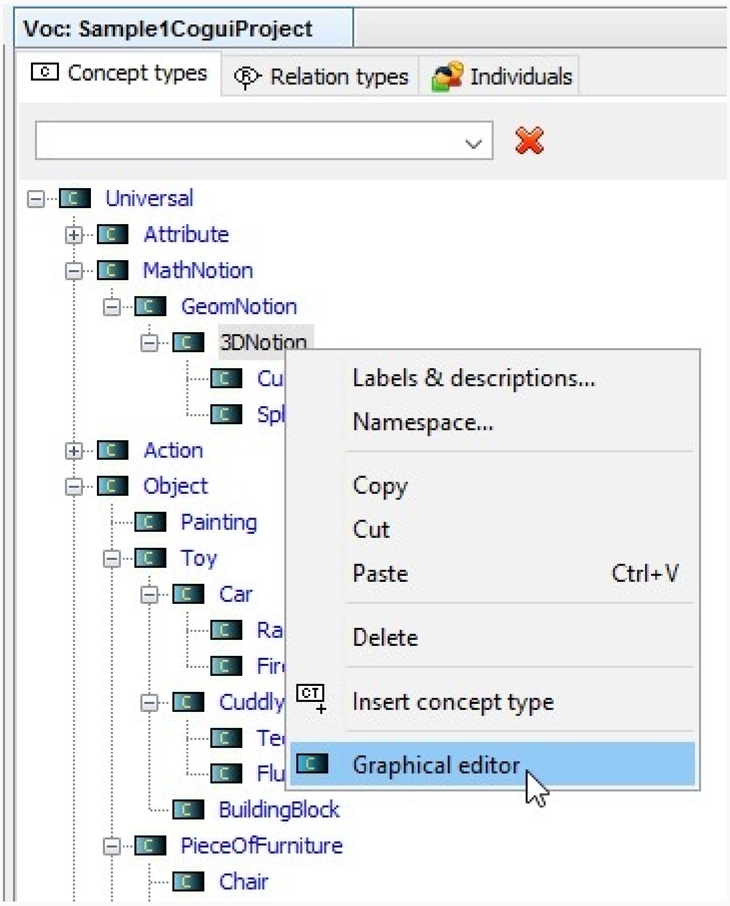


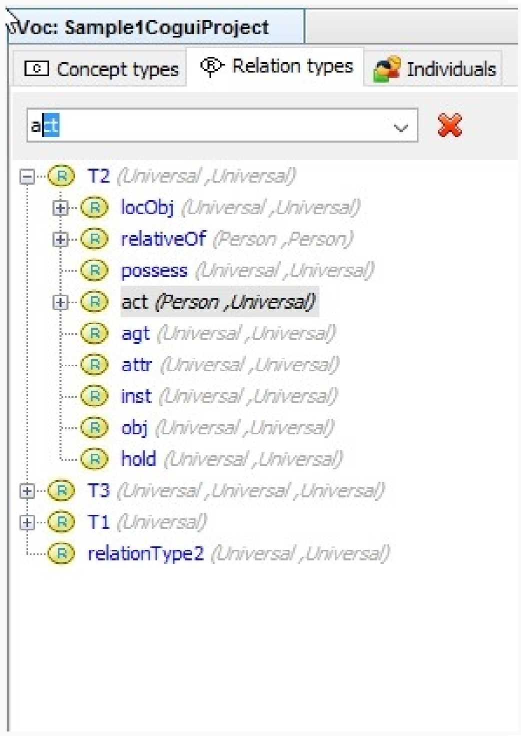
concept types



relation types

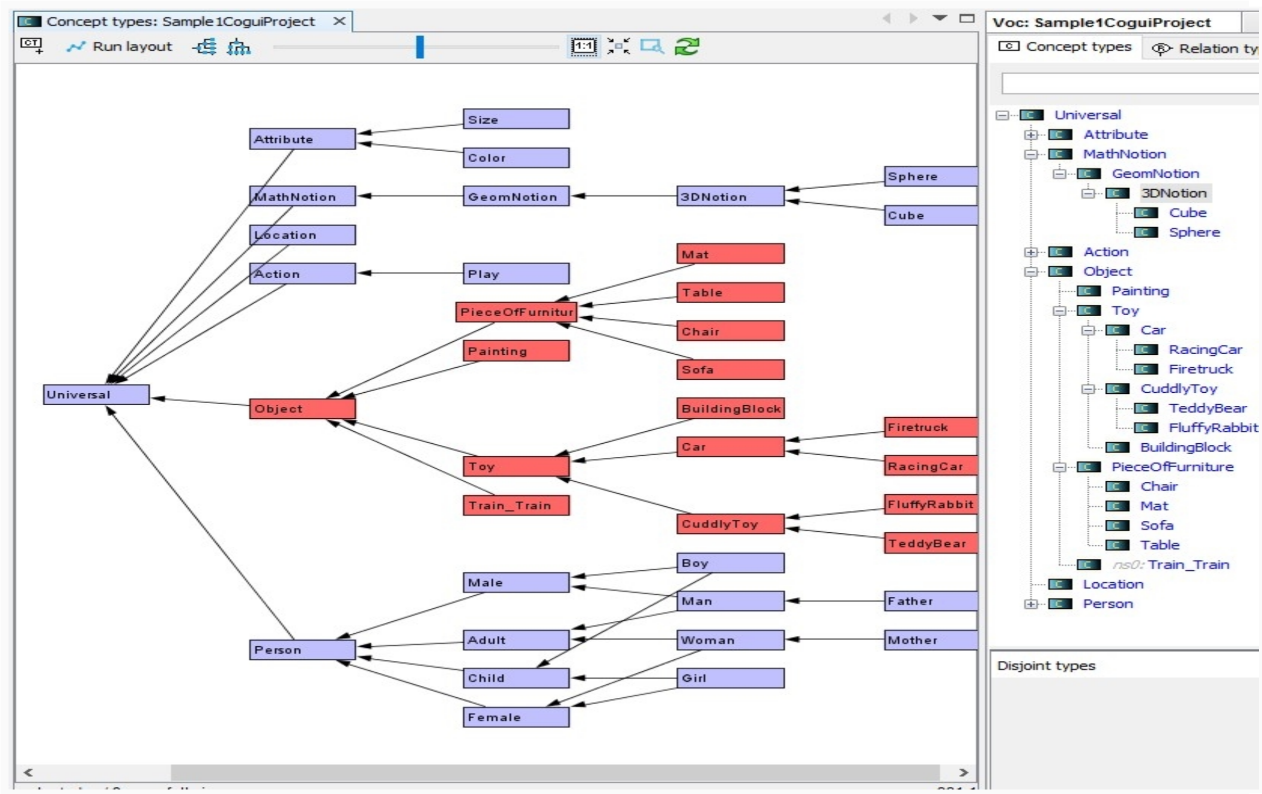
The tree representation is useful to create vertices in conceptual graphs by dragging types into the graph editor (see Graph Edition chapter). Please remember that the type's order is not necessarily a tree. That's why the same type may be retrieved several times in the tree representation. For the same reasons tree is not automatically expandable if hierarchy contains at least one circuit.





Click right button and choose 'Graphical Editor'

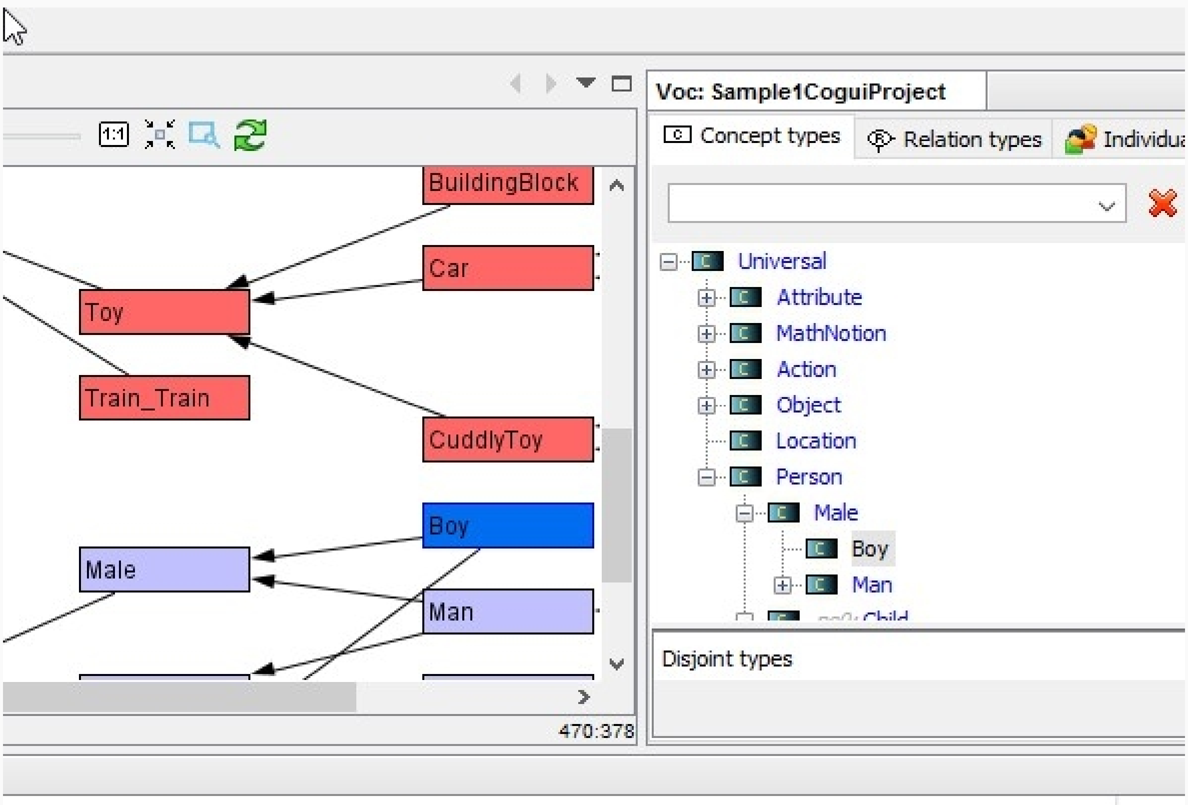
The graphical editor is opened



two synchronized editors for a same type hierarchy (both concepts and relations)

you can navigate between both representations:

- a simple double click on a vertex in left panel select and show the (unique) corresponding vertex in the hierarchy view
- a right click on vertex displays a popup menu: the 'Navigate/Show type in tree' action selects (and scrolls if necessary) the corresponding node(s) int the left panel.



Double click will open graphical editor and scroll to make the vertex visible

From the graphical editor to the tree editor

Two other options show parents or children inside the graph (the scrolling process is automatically performed).

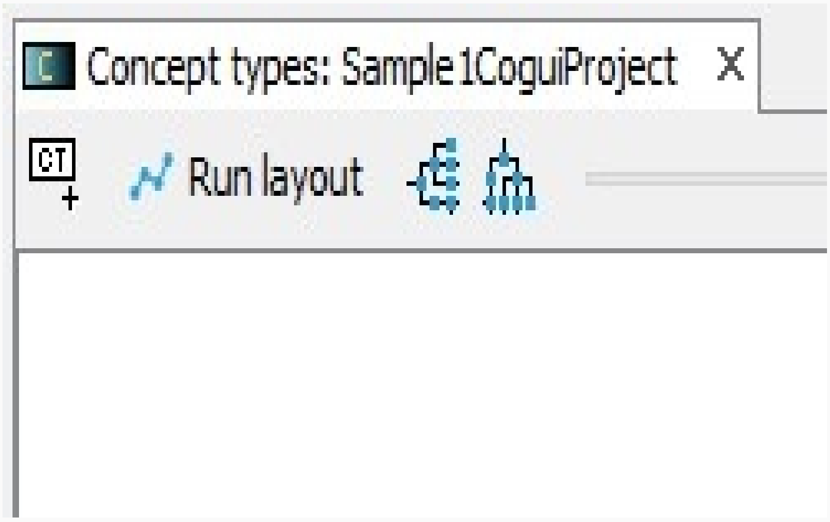
- Shows parent vertex in the graph representation and select them
- Shows children vertex in the graph representation and selected them

Concept types hierarchy

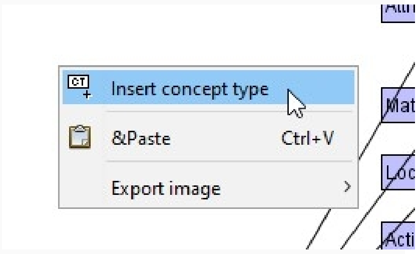
- Insertion
- Graph arrangement
- Concept type hierarchy control
- Forbidden types
- Concept type alteration

Insert new concept type

A newly created concept type hierarchy contains only one type named 'Top' .

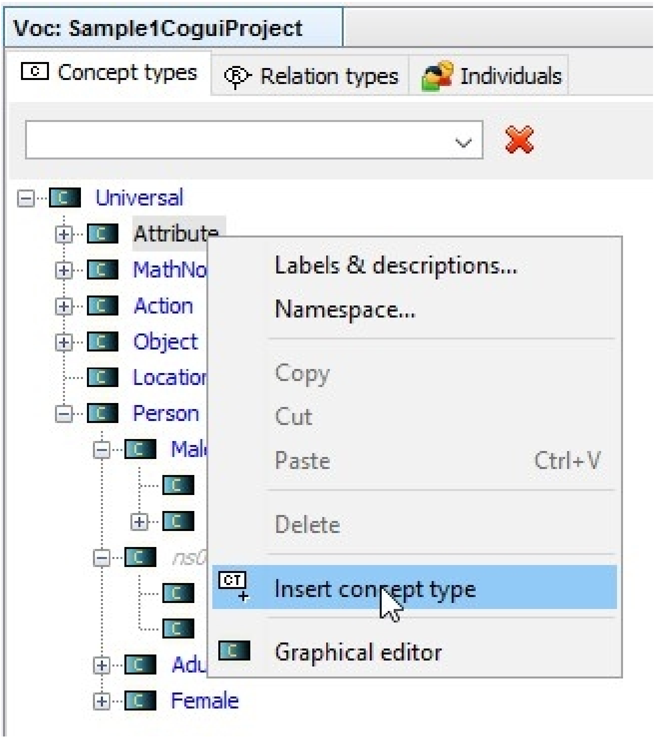


insert button on the toolbar of the graphical editor can be used

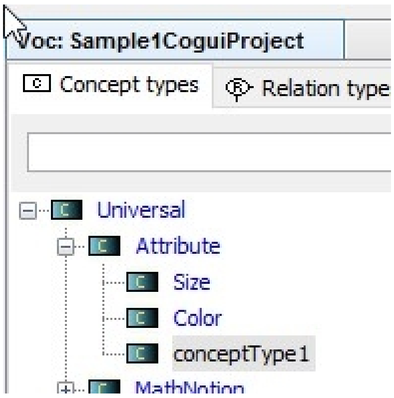


'Insert concept type' into the popup menu inserts the desired type at the mouse location.

A concept type can also be created into the tree representation. 'New concept type' option in the type view popup menu creates a new concept type as a type of selected item:



A concept type can be created directly into the tree

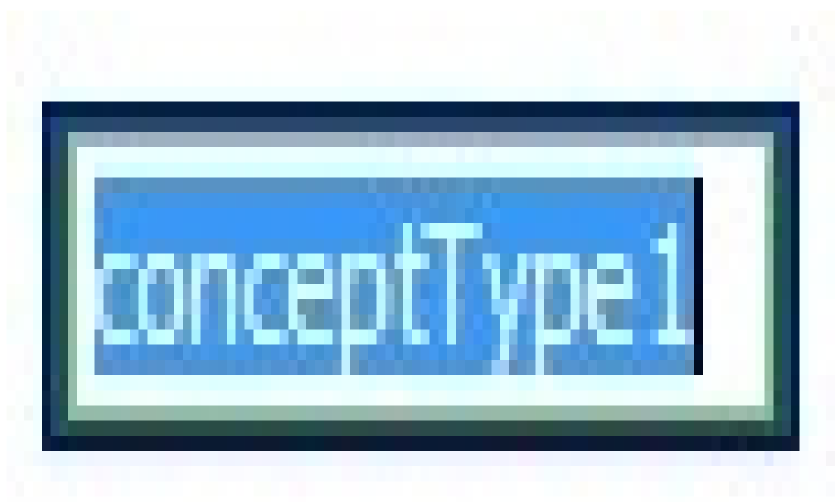


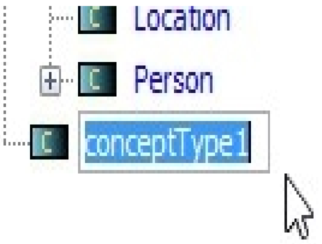
Newly created type is kind of previously selected item

Into the graph editor, the following vertex is displayed: 

. Click twice on the vertex to edit

type name. Concept types can be renamed directly on the concept type tree. Click once on the tree item to edit type name. Both actions have same effect and are synchronized.





Rename on graphical editor

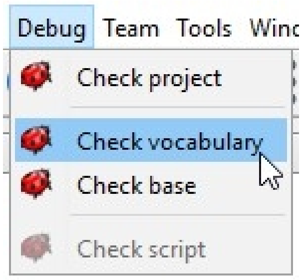
Rename on tree editor

Naming convention

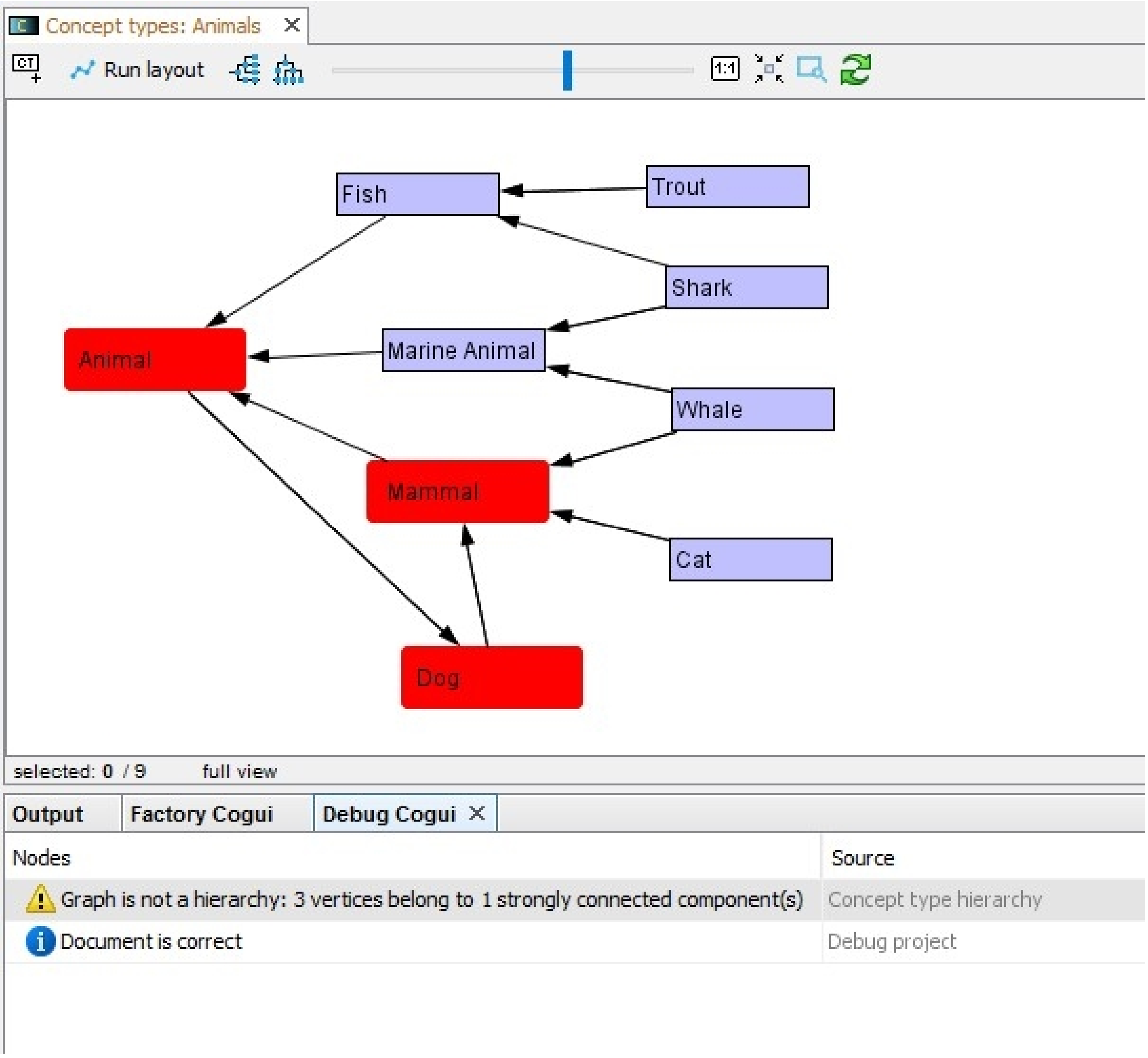
Homonymous types are not allowed in the same type hierarchy. The case is respected but comparisons are case unsensitive. For instance user can decide to write 'Dog' or 'dog' but cannot define both words in the concept type hierarchy. Blank spaces are allowed.

Concept type hierarchy control

Action to control of the concept types hierarchy is provided in Debug menu:



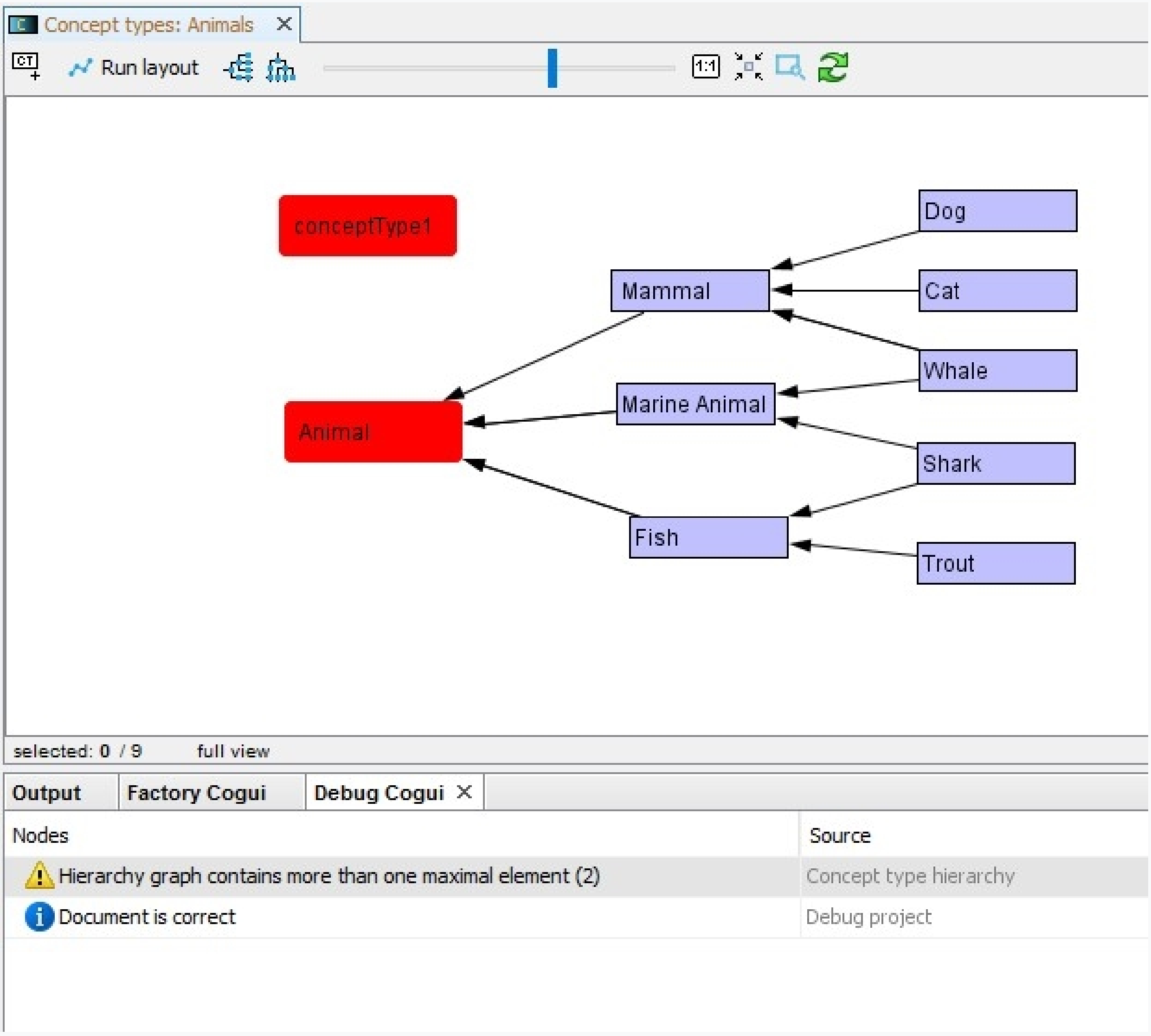
Only one critical error can occur with the graph structure: the detection of a circuit. Assume that 2 types A and B on a circuit, i.e. a path exist from A to B and another exist from B to A. It means A is a kind of B and B is a kind of A



CoGui control detects circuits: all the animals are not dogs

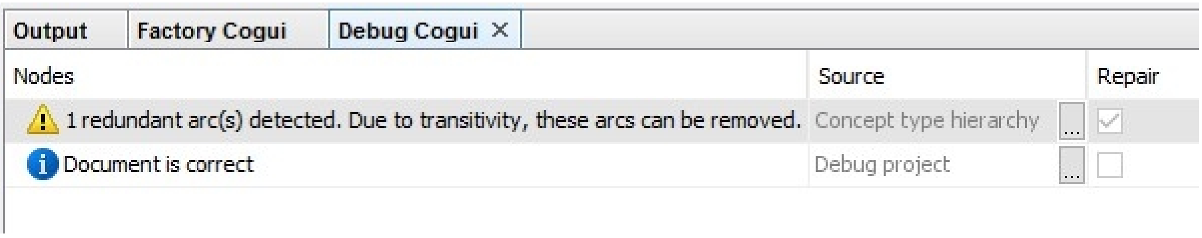
However, it is possible for a project consisting of several pieces of ontologies to work with several synonymous types from multiple equivalent URIs (owl: SameAs). Despite the warnings CoGui is able to work with the circuits, all the concept types belonging to the same circuit are considered as equivalent.

Another model constraint is that the concept types hierarchy must have a maximal concept type. By default CoGui names it 'Top', feel free to change its name or to choose another vertex as the maximal. A warning message occurs if the hierarchy contains more than one maximal element. The tool does not automatically add a maximal abstract type to the hierarchy but it is recommended to respect this constraint.

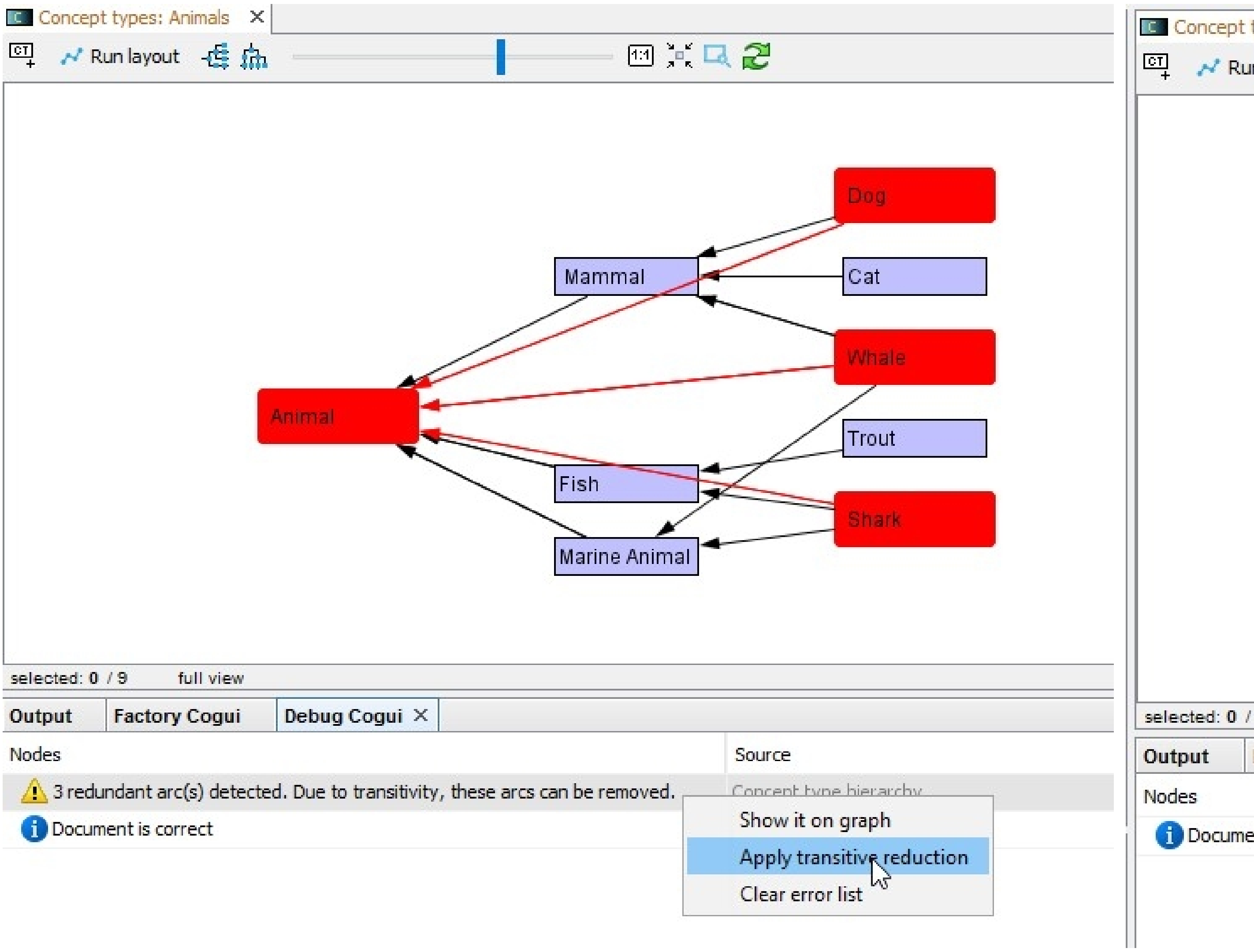


A single maximal type is required

Another warning can occur when the user draws redundant arcs. If A is a kind of B and B a kind of C, by transitivity CoGui 'knows' that A is a kind of C, hence an arc between A and C is correct but redundant. These extra arcs can obstruct the graph view but extra entries on the tree representation could be used as sort of shortcuts for often used types. That is the reason why CoGui accepts and stores redundant arcs. When a message (error or warning) occurs, it can hold the action to solve it. The Repair box checked indicates that a repair action is available:

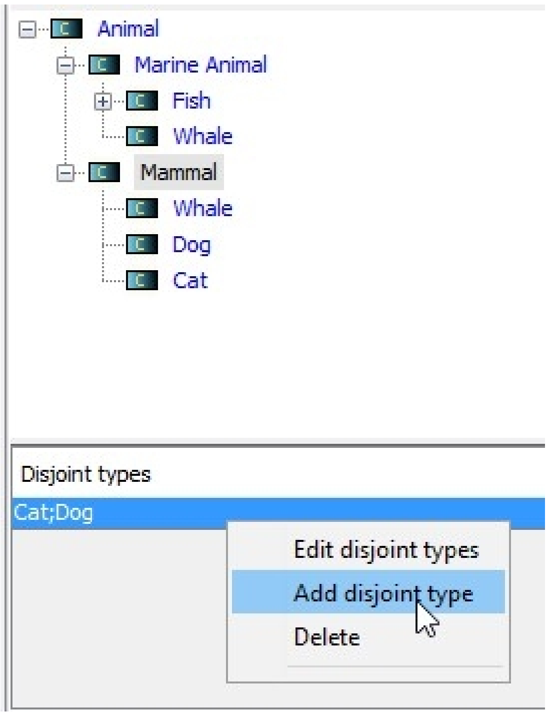


In this case all the redundant edges can be removed with the message popup menu with the action named 'Transitive reduction'.

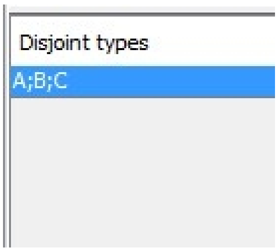


Forbidden types

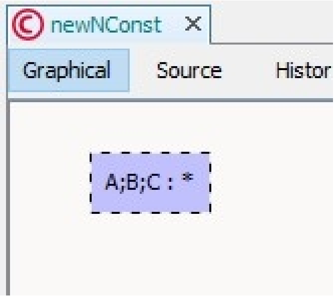
In a graph, the concept vertices may be associated to a conjunctive types, meaning it has several types. As a result, the model provides a mechanism to prohibit some incongruous associations. For instance, suppose you have defined both "Animal" and "Plant" concept types, you might want to prohibit associations between these types as well as between sub-types of them. It is possible to express this restriction in your concept type hierarchy. To this end, you are going to introduce a forbidden type in the concept type View to express this incompatibility. See below an example of such restriction expressed in the concept type hierarchy triggers and here it triggers an error in a conceptual graph.
Forbidden types (also named disjoint types) can be added in the view placed below the concept type tree:



It is possible to create sets of 3 conjunctive types or more. If the forbidden type (A;B;C) is defined, all subset will be forbidden (A;B) (B;C) (A;C) and of course (A;B;C). This corresponds to the most frequent needs of the users. If you want to specify that only the conjunction (A;B;C) is forbidden a negative constraint can be used.



All subsets of A,B,C with card>1 is forbidden



Negative constraint for A;B;C

Concept type alteration

As for relations, adding concept types does not affect the existing ones. The deletion of a concept type can affect not only the type hierarchies and fact graphs, but also the signatures of relations. All references to this type must first be removed from the base.

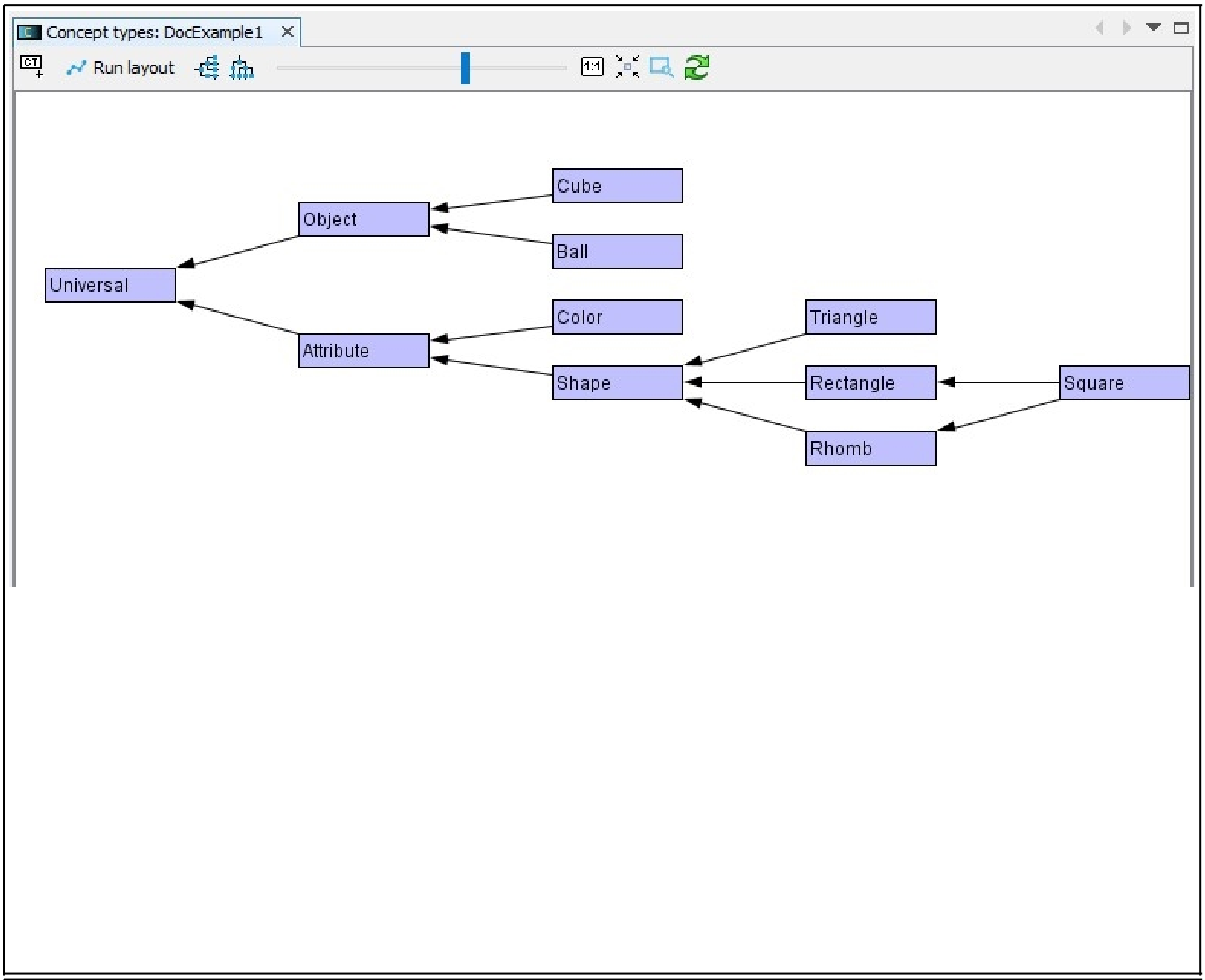
Removing a link between two concept types does not create inconsistencies in the knowledge base but can decrease the number of answers to a query, adding a link can increase the number of answers, the forbidden types may change and some constraints may become unsatisfied.

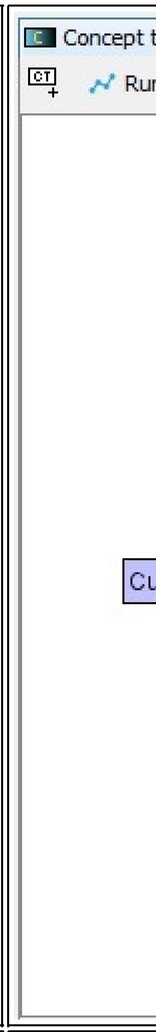
All consequences of these alterations are detected by the CoGui controller and error messages help the user to correct inconsistencies.

Graph layout and coloring

Even if the position and color of the vertices of the graph do not matter in the representation model, they can be very useful for the user.

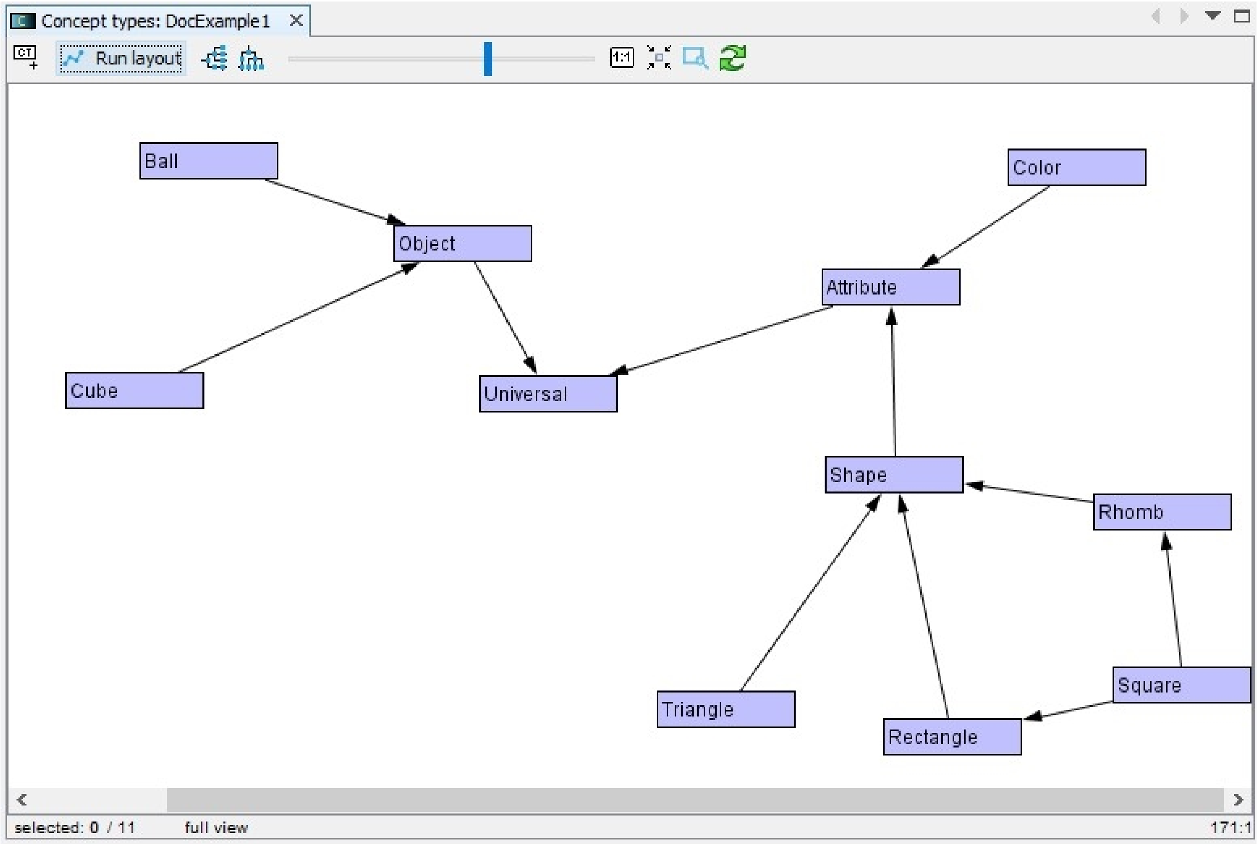
A vertex is moved by dragging its box with the mouse. Another way to place vertices is to run an automatic arrangement with the layout algorithms.



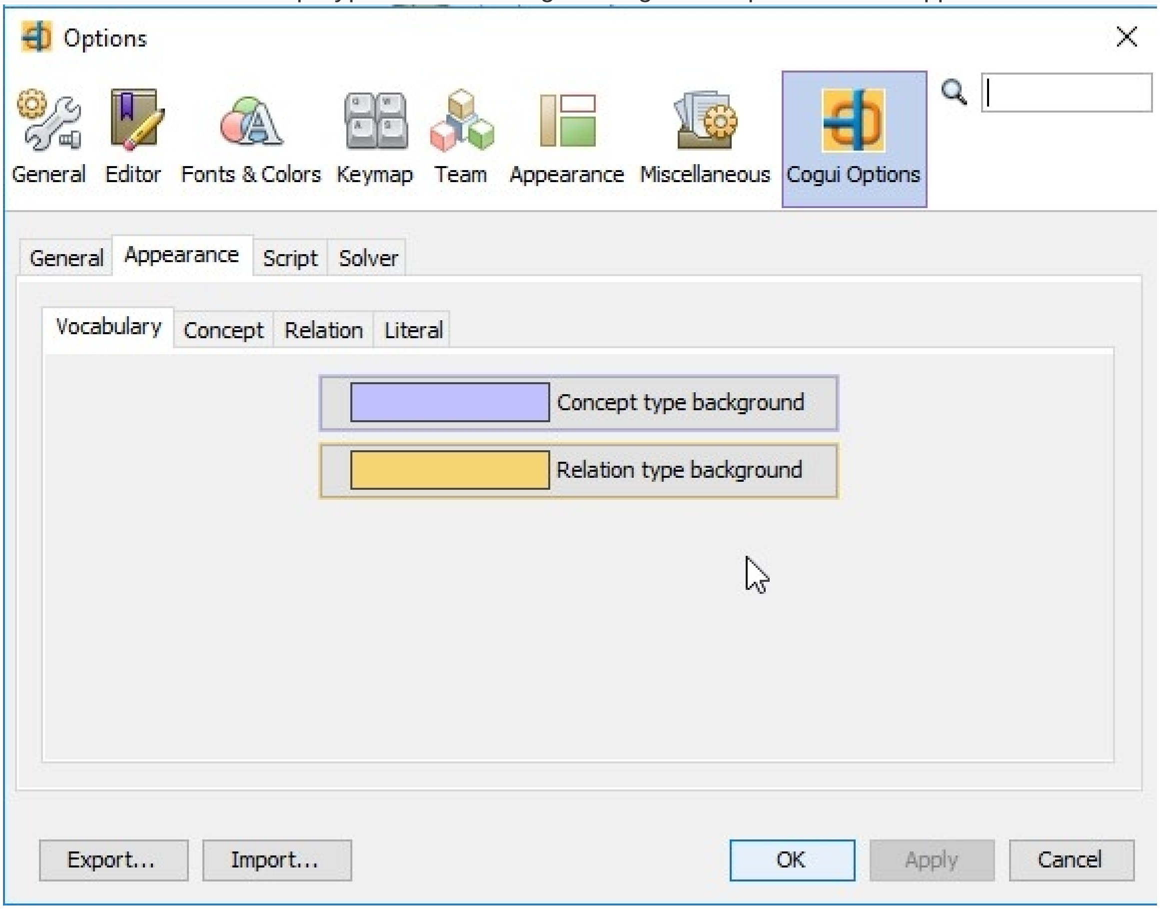


horizontal tree layout

A dynamic force directed layout is also provided:

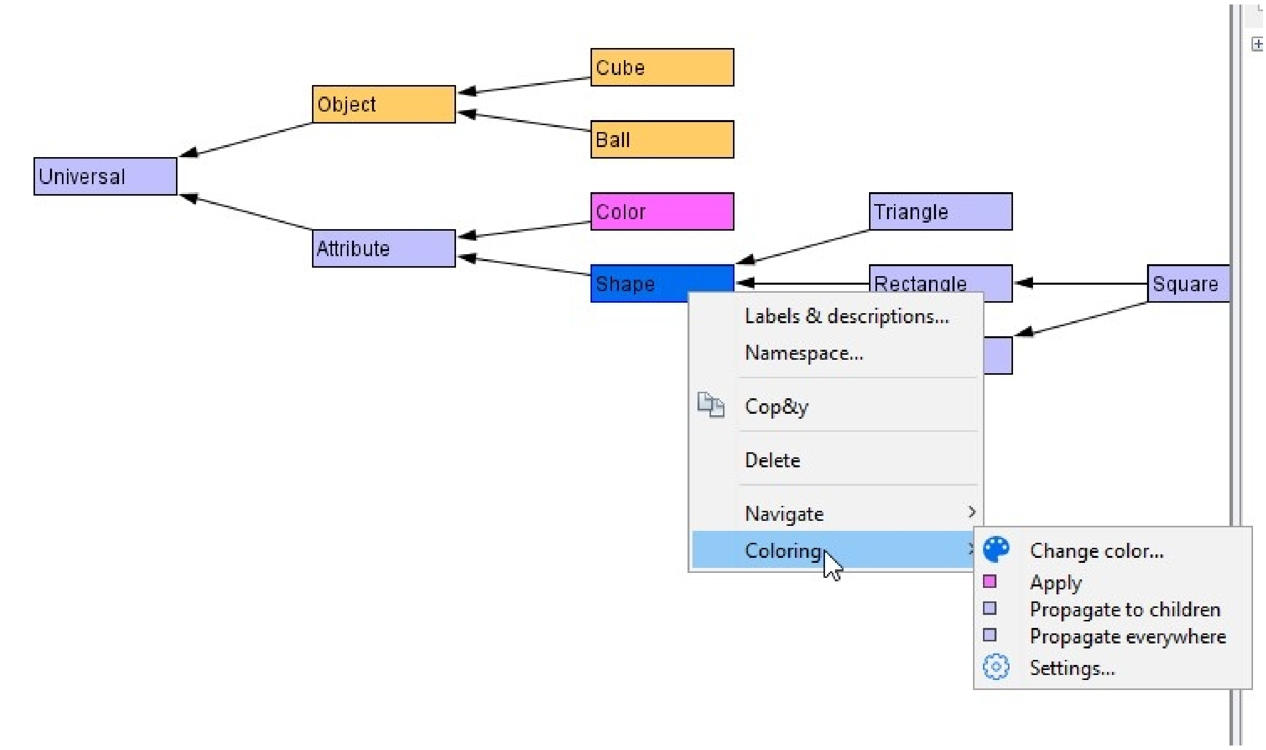


The default color of the concept types can be changed using Tools/Options/CoGui/Appearance command:



A different color can be selected for each concept type. Contextual menu of the vertex propose a submenu

'Coloring':

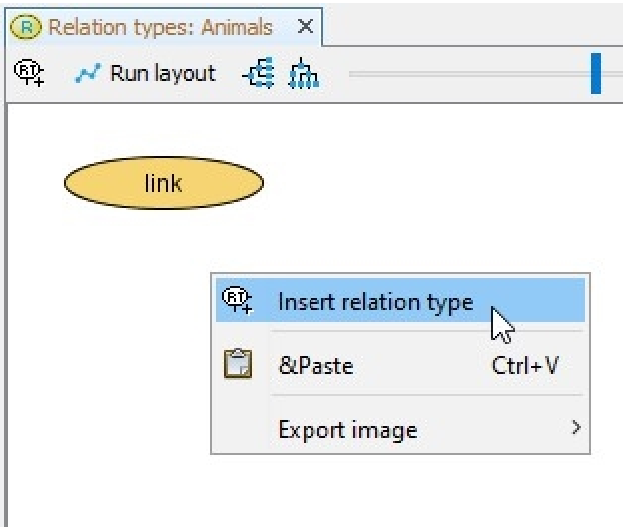


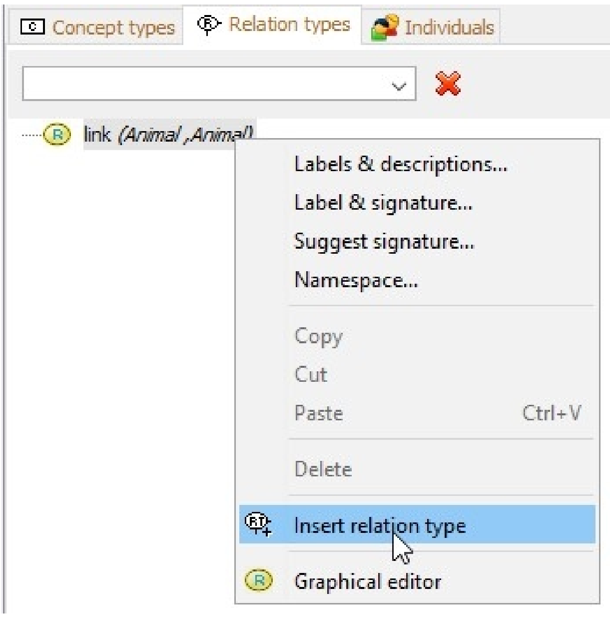
Relation types hierarchy

- Insert new relation type
- Relation type signatures
- Graph layout and coloring
- Relation type hierarchy control
- Relation type alteration

Insert new relation type

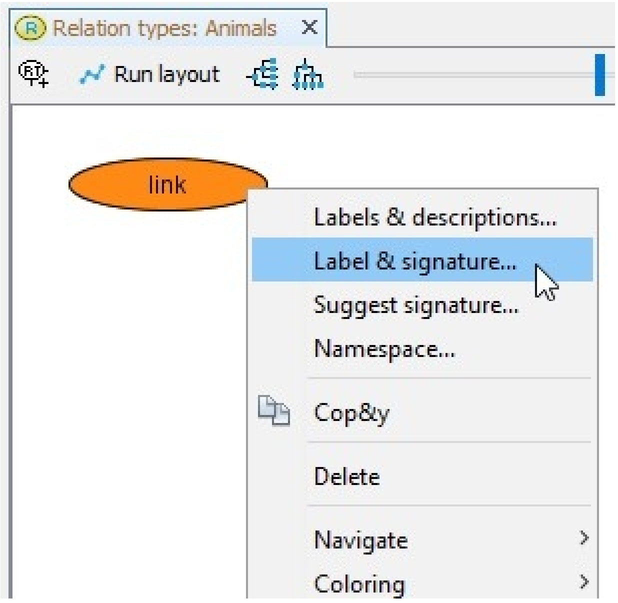
A newly created relation type hierarchy contains only one type named 'Link' .

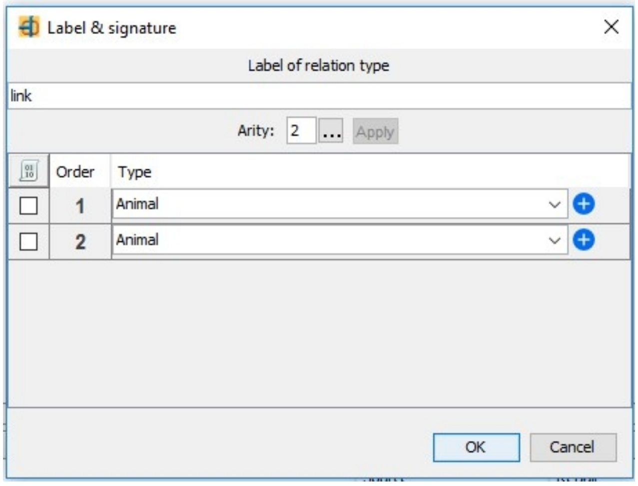




Use button or popup menu to add a new relation type A relation type can be created directly into the tree

The relation type can be updated with 'Label & signature...' menu action:



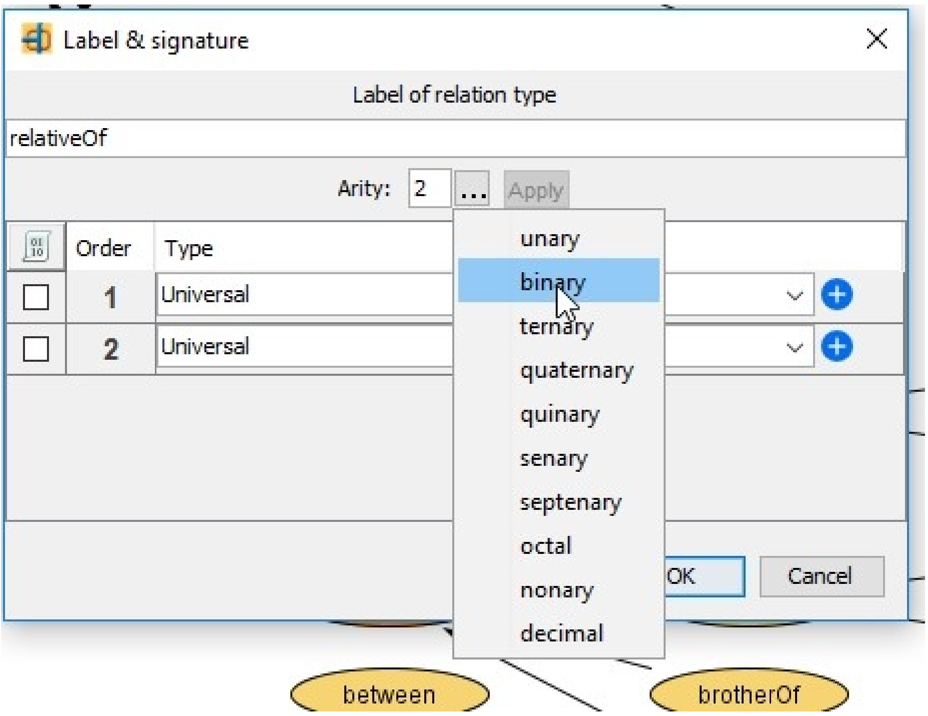


The menu action The dialog box

Homonym types are not allowed in the same type hierarchy. The case is respected but comparisons are case insensitive. For instance the user can decide to write 'Binary_rel' or 'binary_rel' but cannot define both words in the relation type hierarchy. Blank spaces are allowed.

A signature must be associated with each relation type. A signature is an ordered list of concept types (numbered from 1 to arity) where arity denotes the arity of the relation type, i.e. its number of arguments. The signature dialog box allows to change the arity and to specialize involved concept types.

Press the assistant button and choose arity or directly edit arity number and press the 'Apply' button to confirm. Lines are added or removed from the table. Each concept type can be changed directly or with the assistance button:

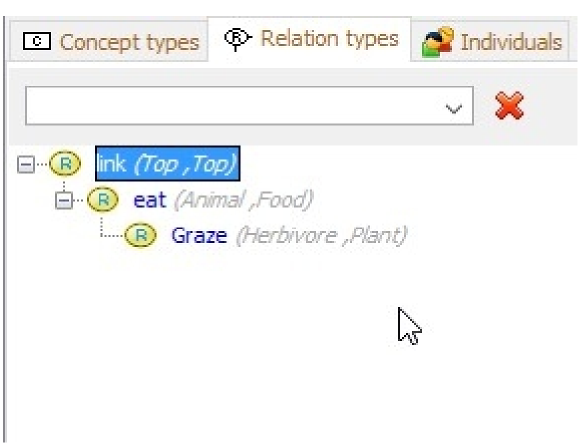


Relation type hierarchy control

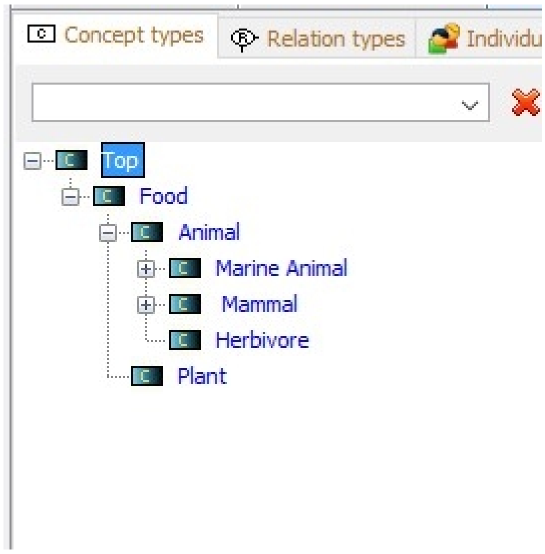
Similar to concept type hierarchy, circuits are forbidden. The only difference with concept type hierarchy is due to signatures. The constraints are:

1) relation types are grouped by arity. Each 'arity family' must have a maximal element. It means that the relation type hierarchy is decomposed w.r.t. the arity and a unique maximal element is required for each of these sub-hierarchies.

2) Let A and B be two relation types in the same sub-hierarchy (i.e. A and B have the same arity). If A is a kind of B, it means that every concept type in A signature is respectively a kind of concept type in B signature. For example if graze(herbivore,plant) is kind of eat(animal,food) their signatures respect compatibility if herbivore is a kind of animal and if plant is a kind of food.

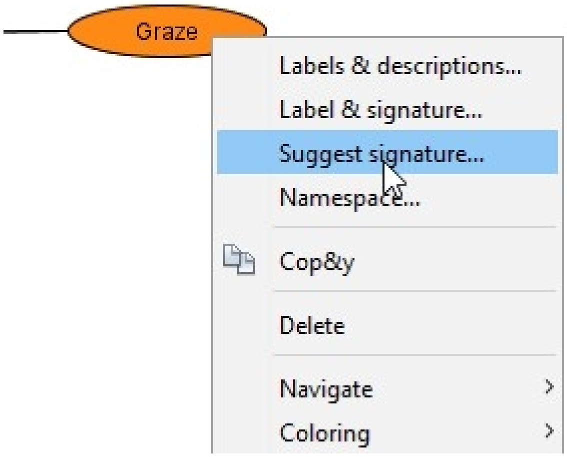


graze(herbivore,plant) is kind of eat(animal,food)



herbivore is a kind of animal and plant is a kind of food

Another way to quickly define or pre-define signatures is to use 'Suggest signature...' command on popup menu. Use the command after a link is established between a new relation type and an immediate greater relation type: the parent signature is automatically proposed. But this command is designed for more a complex purpose.In a complex ontology it becomes difficult to define a new relation type signature. The command 'Suggest signature...' can help to find the maximal compatible signature.



Relation type alteration

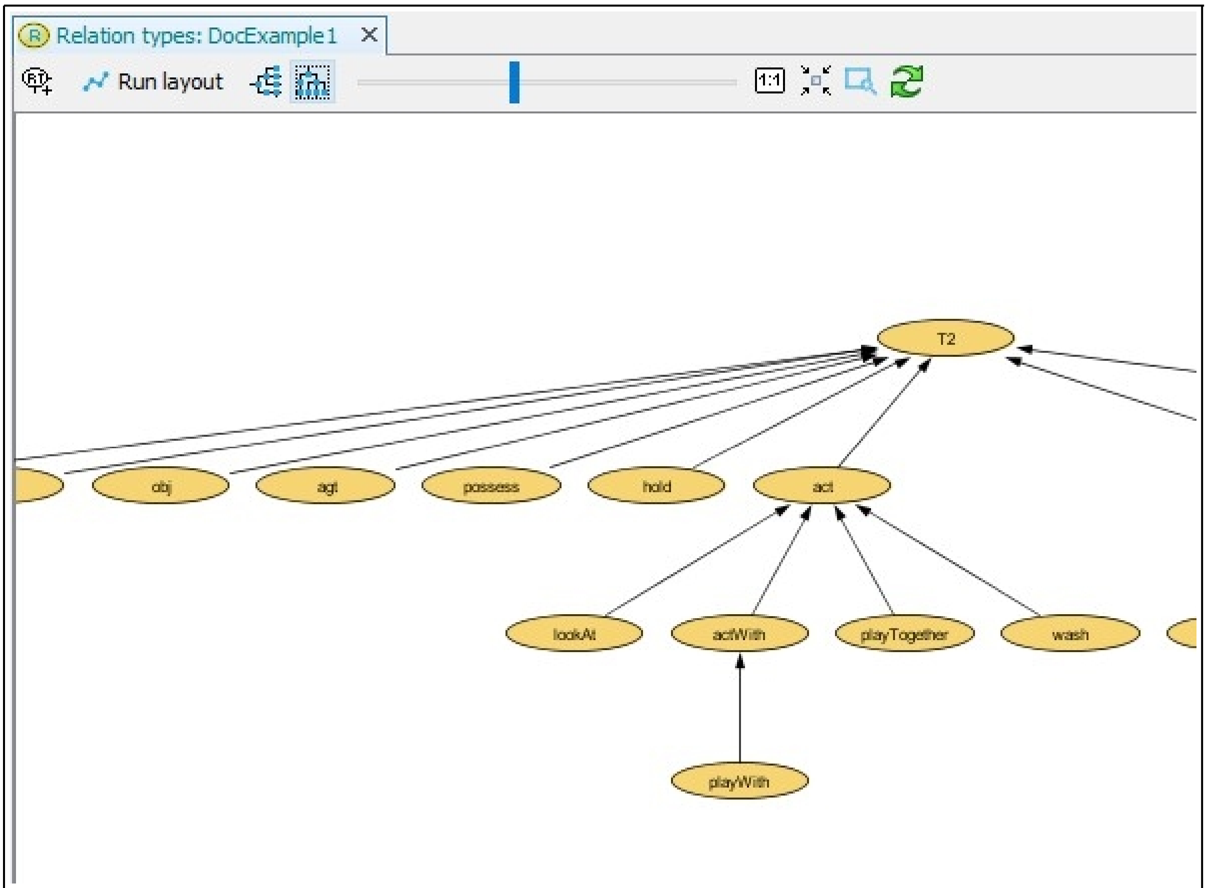
Relation type labels can be changed. With respect to the signature covariance new relations can also be added without consequences for existing knowledge.
For obvious reasons of referential integrity, the removal of a relation assumes that all occurrences have disappeared from all graphs, both in annotations and within ontology.
The consequences of the change of a signature depends on its nature: if the arity of the signature is changed, all occurrences of the relation will require user’s intervention; if only the concept types of the signature are changed, then it will be a different signature.
If a concept type is replaced by a more general type, the content of the knowledge base will not be affected, and no error will occur. However, if a term is specialized, it can have an effect on the content of the knowledge base, and can also trigger errors in the annotations. Removing a link between two relation types does not create inconsistencies in the database but can decrease the number of answers; adding a link can

increase the number of answers (new rules may be applicable) and some constraints may become unsatisfied.

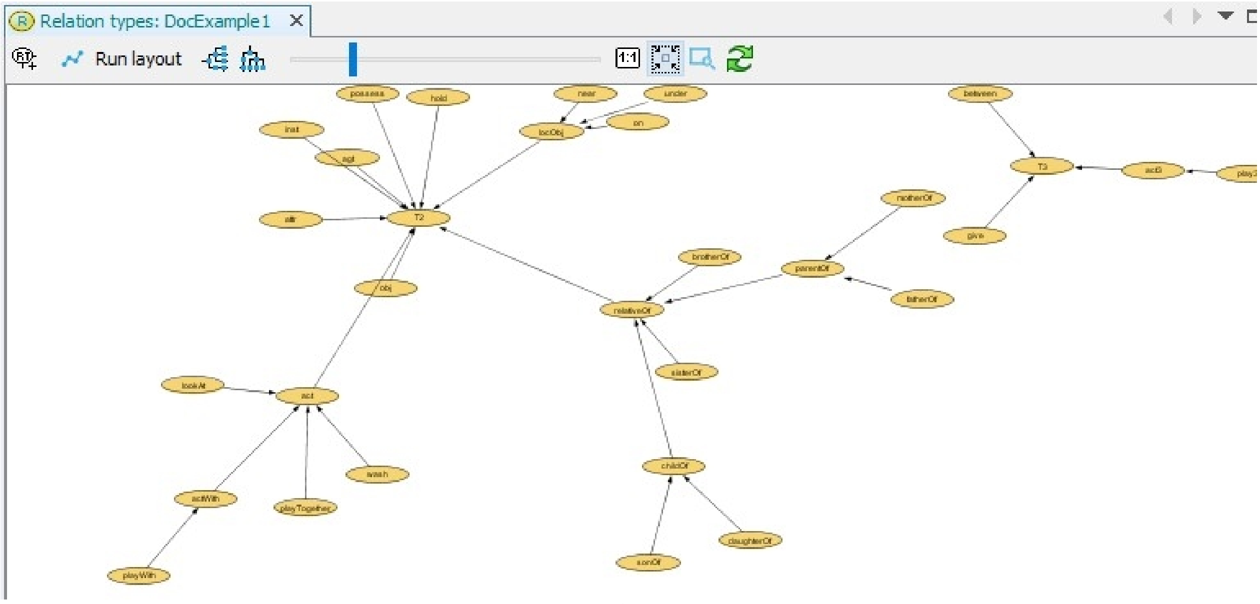
Graph layout and coloring

Even if the position and color of the vertices of the graph do not matter in the representation model, they can be very useful for the user.

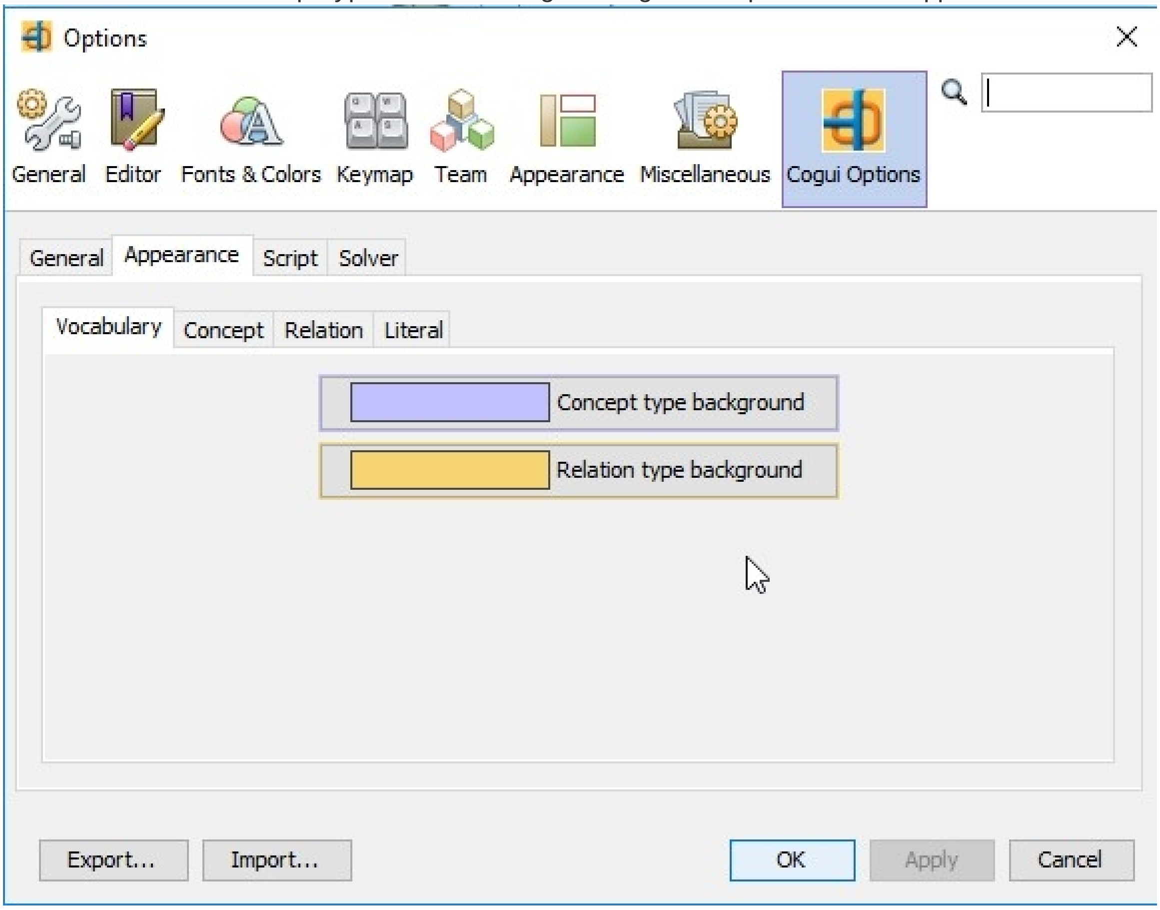
A vertex is moved by dragging its box with the mouse. Another way to place vertices is to run an automatic arrangement with the layout algorithms.

	
horizontal tree layout	vertical tree layout

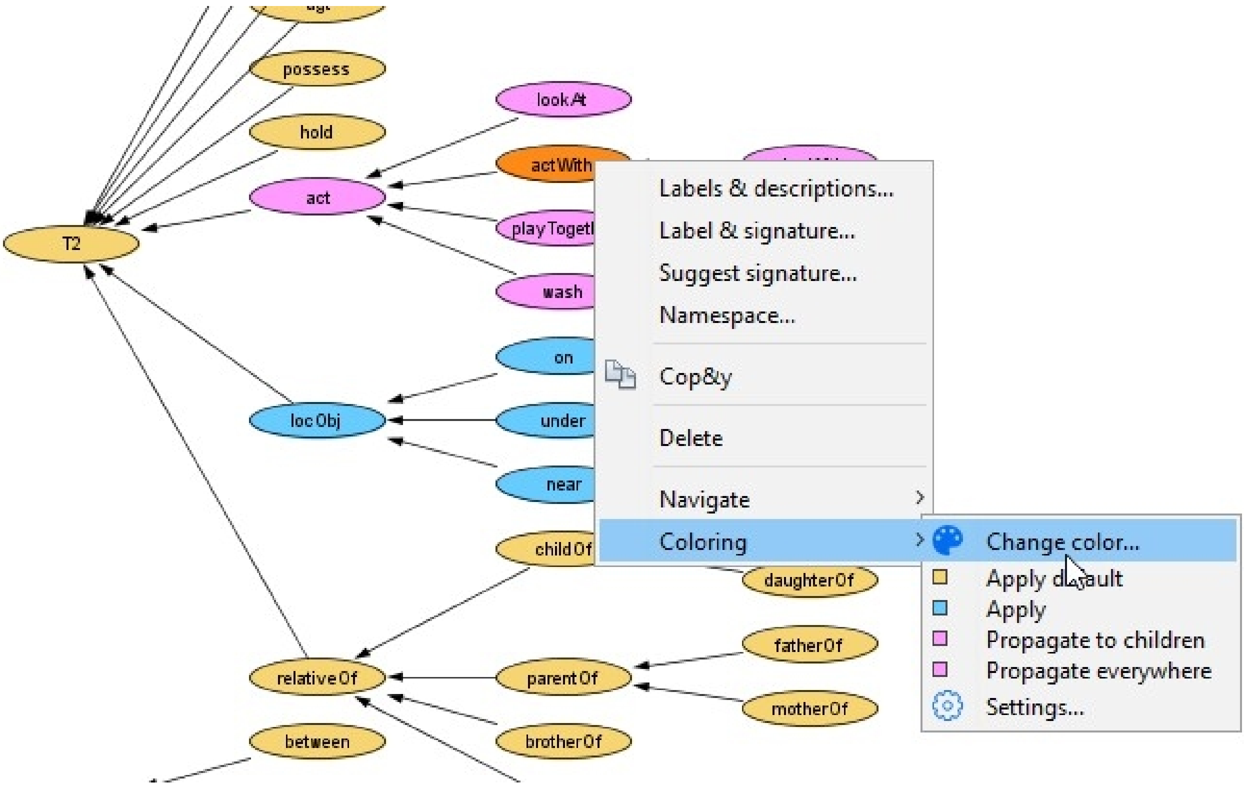
A dynamic force directed layout is also provided:



The default color of the concept types can be changed using Tools/Options/CoGui/Appearance command:



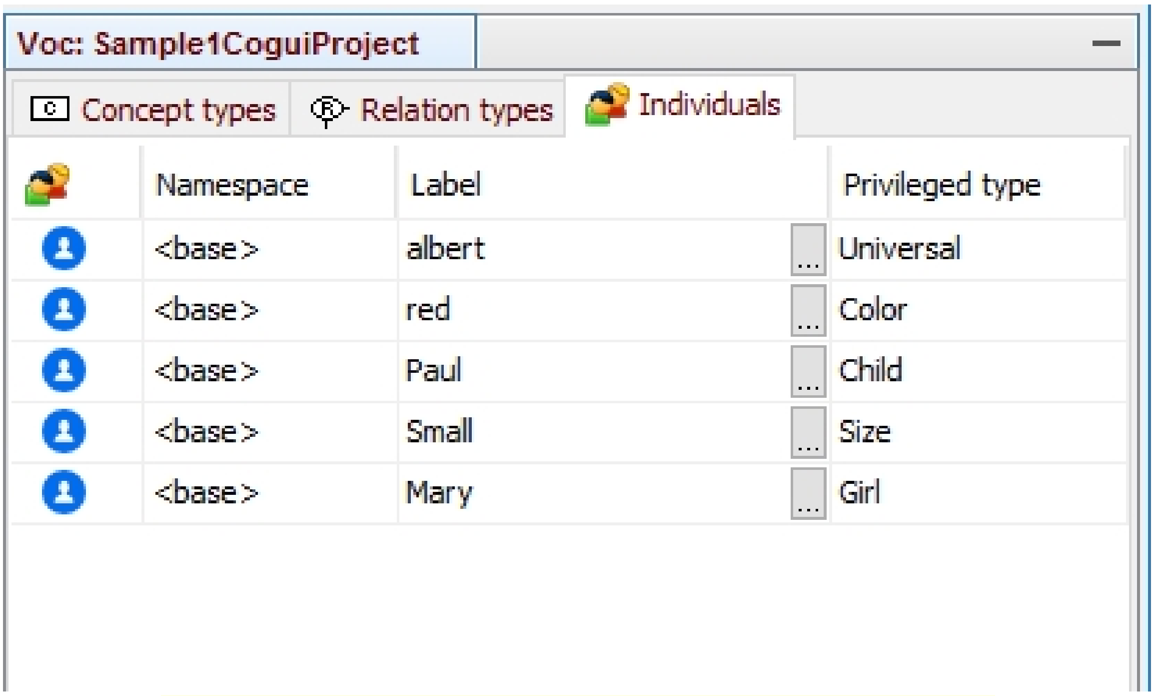
A different color can be selected for each concept type. Contextual menu of the vertex propose a sub-menu 'Coloring':



Individuals

An individual is an identifier which is a surrogate for a precise entity of the discourse universe. For instance, if Town is a concept type then Budapest is an individual of type Town. A concept type may have subtypes, e.g. SmallTown could be a sub-type of Town, an individual cannot have 'sub-individuals'. The ontological individuals are the individuals about which all the users agree, i.e. for all users an ontological individual must represent the same entity in the discourse universe. An ontological individual is entered into a COGUI-ontology with a primitive concept type called its privileged type. For instance, if the COGUI-ontology concerns Modern Art, and if Picasso is an ontological individual of privileged type Artist representing the famous artist Pablo Picasso, then it is impossible to use the identifier Picasso for representing a Citroën car (unless the conjunctive type Car, Painter is not forbidden). COGUI checks that an ontological individual appearing in an annotation has a type which is compatible (i.e. not forbidden) with the privileged type of the individual.
All individuals appearing in a COGUI-ontology (e.g. in rules, constraints or prototypical knowledge) must be ontological individuals. Thus, the set of ontological individuals can be completed only whenever all knowledge representing in a COGUI-ontology have been considered.

The individual view lists all individuals in a sortable table. Needless to complete, the list of individuals automatically updates when the user references individuals in different graphs. Select individuals in the list to drop them to the graphs and right click to popup the contextual menu to rename, change the privileged type.



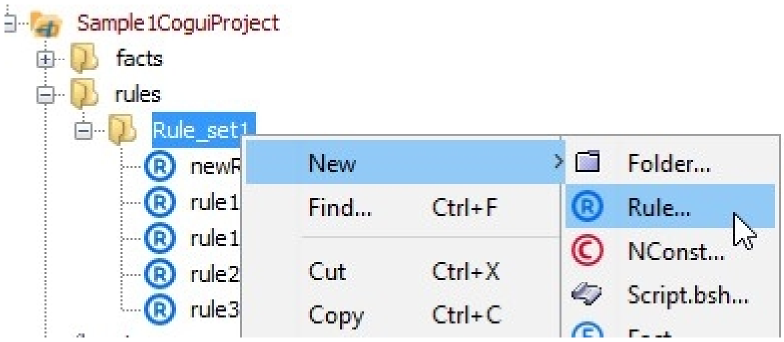
Individuals tab in vocabulary view displays the complete list of individuals

Since 3.0 CoGui integrates the notion of namespace. Then a namespace can be associated the each individuals. It can be selected directly on the list:

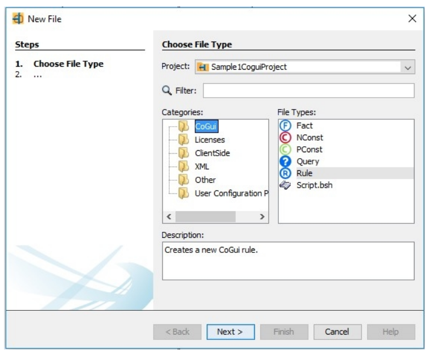
Rules

To create a new rule or edit an existing rule go on the projects view and use the popup menu (right-click). As other graphs, rules can be organized in folders. It is particularly interesting to classify the rules on families, especially for testing purposes. Editing rules is very similar to editing Facts. But the split editor is divided in two parts: hypothesis and conclusion. By default hypothesis is placed on the left part and conclusions on right part of the editor. The split bar can be oriented with the mouse to horizontal position, in this case hypothesis is placed on top and bottom is reserved for conclusion.

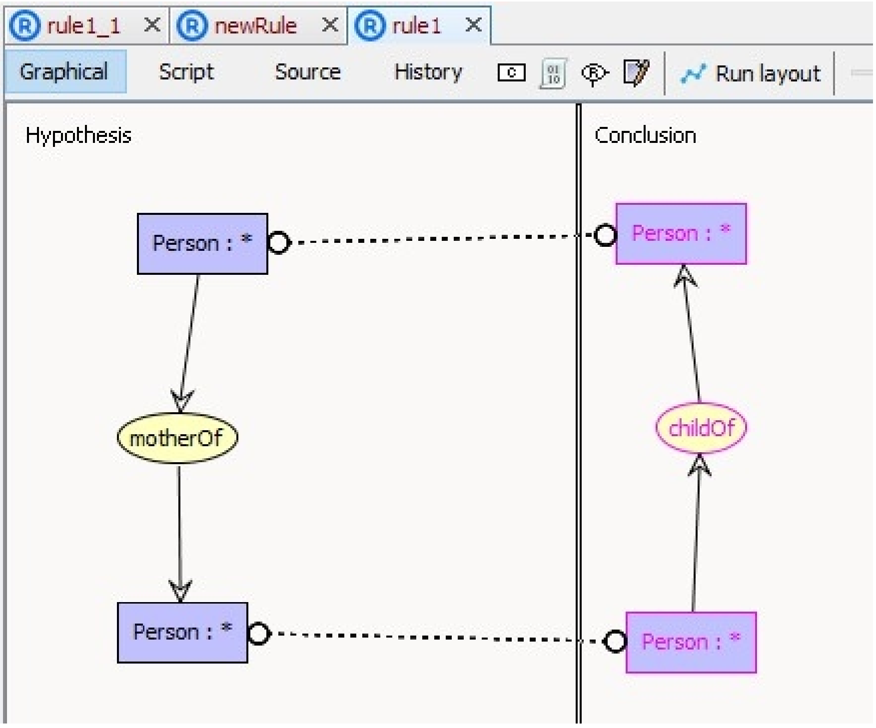
Create a new rule:



If "Rule..." action does not already appears in menu choose "Other..." and select the type of CoGui object that you want to add in your project:

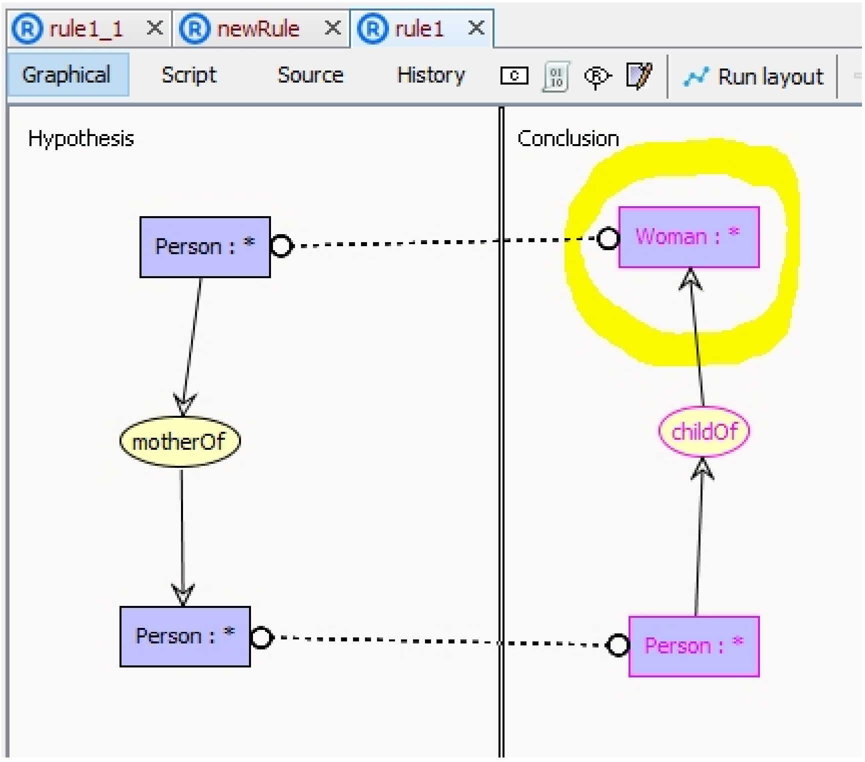


Rules are used to represent implicit (common sense) knowledge. For instance, let us assume that the fact that Eve is the mother of Abel is represented in a fact graph. If the ontology contains a rule saying that **if x is the mother of y then y is a child of x** then the system can automatically add the information that Abel is a child of Eve.
Such a rule is represented by two simple graphs. One represents the hypothesis (e.g. [woman]-1-(mother of)-2-[human]) the other represents the conclusion ([human]-1-(child of)-2-[woman]). Furthermore, there is a link between the first person in the hypothesis and the second person in the conclusion, and a link between the second person in the hypothesis and the first person in the conclusion. A rule "if A then B" is used as follows: if an annotation contains A then B can be added to the graph. See below the graphical representation of this rule:



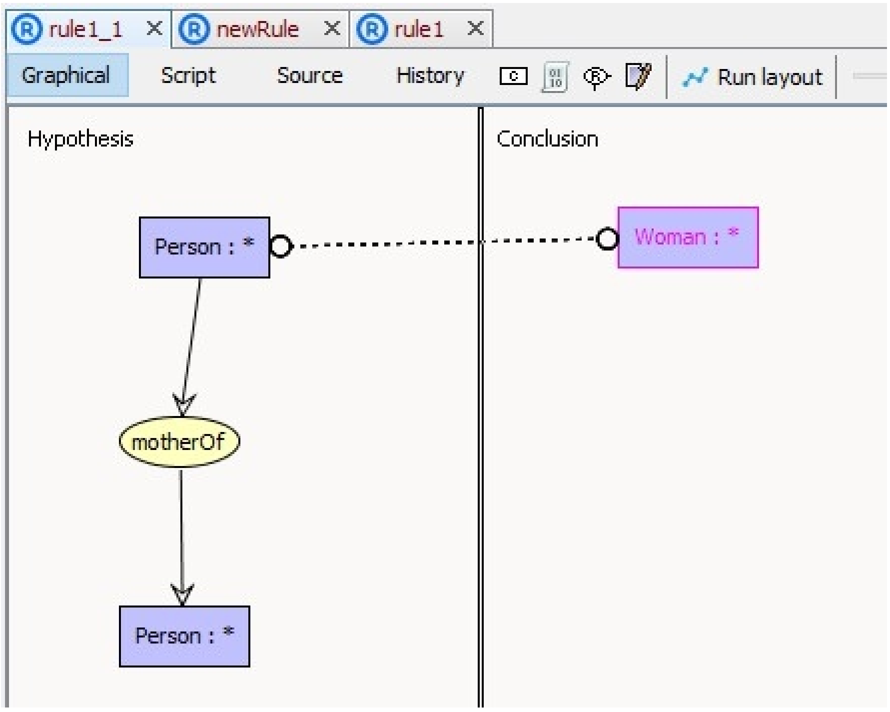
A rule example

A bi-colored representation of this rule could be simplest with just a conclusion relation "child of" added to the hypothesis. The advantage of this representation is that it allow specialization of a concept type in the conclusion. For instance, from the hypothesis [human]-1-(mother of)-2-[human] it can be deduce that the first person is a woman. Thus, rule represented below is more powerful than previous example:



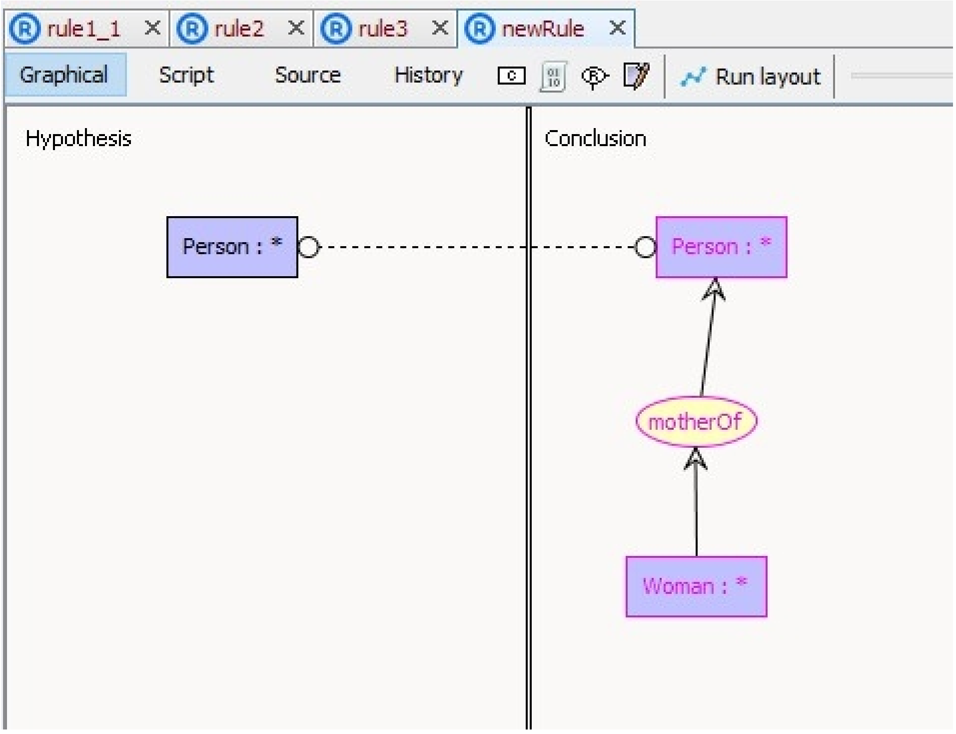
rules could generate some specializations. Here Person is eventually specialized in Woman.

But it is better to express each deduction in a separate rule and add a new rule to the first one:



a simple rule for each deduction

Do not worry about rule applying order, even if the first rule is not applied in a first step, it can be appliable after applying another one. All rules are tried until saturation (when no rule is appliable).
Due to saturation, rules must be built carefully when at least one rule can create new concepts: The rules may loop and cause an infinite production of concepts. Example below is trivial but combinations of rules can generate loops very difficult to detect.



This rule will cause a loop

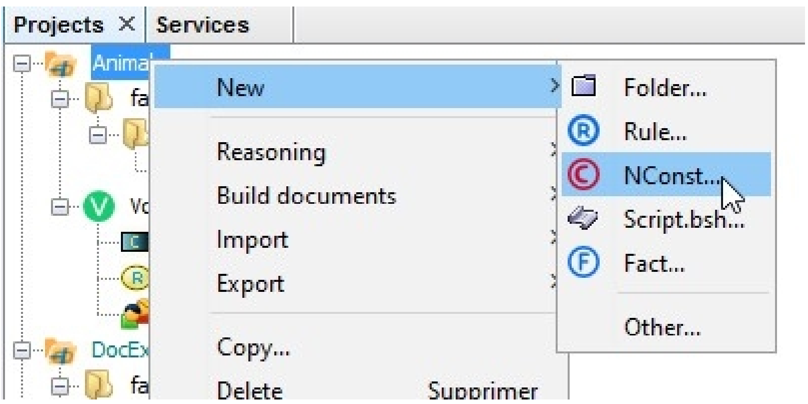
Fortunately it is possible to interrupt the saturation running task if it did not end after a "reasonable" time. And the knowledge base can be queried without using saturation. See Applying rules section and Querying for more about the use of rules.

Constraints

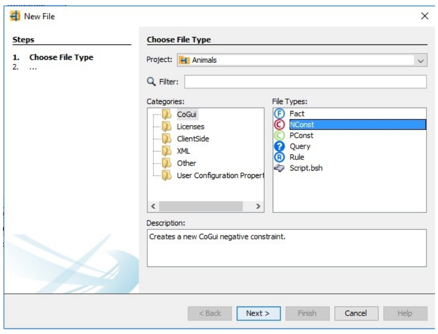
Constraints allow to define pieces of information that are forbidden in the facts (negative constraints) or mandatory in the facts (positive constraints).
To create a new constraint or edit an existing constraint go on the projects view and use the popup menu (right-click).

You will find all the information about the use of the constraints are in section Inspecting facts

Create a new constraint:

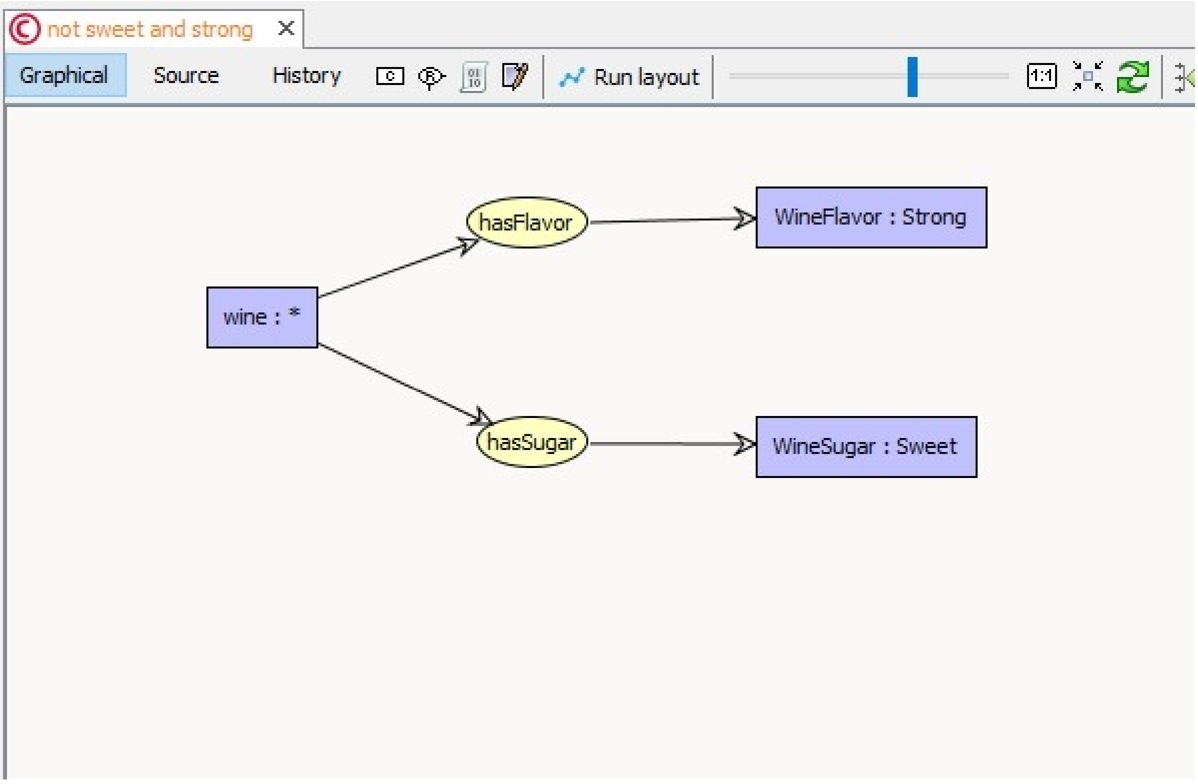


Select the type of Constraint that you want to add in your project:



Negative constraints

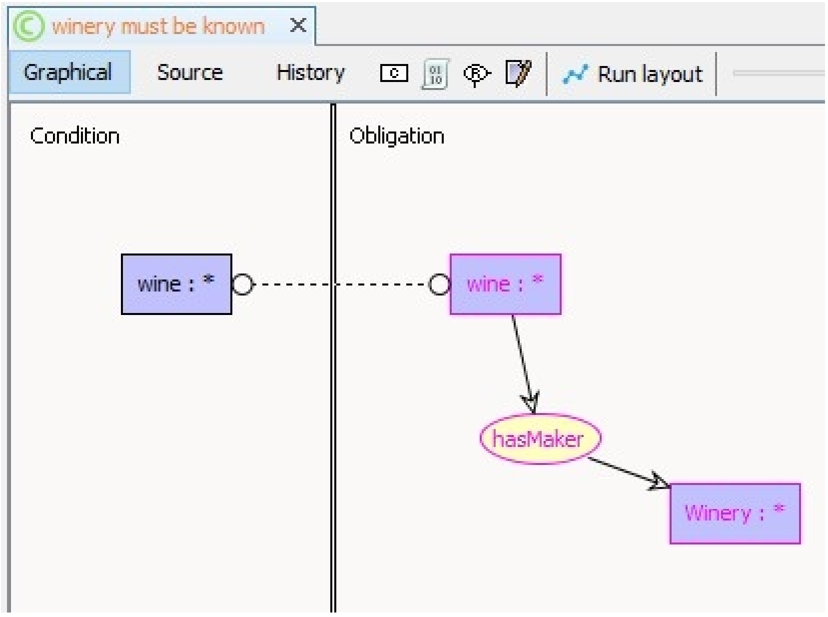
A negative constraint is a simple graph expressing a condition that must not appear in checked facts. Checking a negative constraint is similar to query facts. Facts are validated if no homomorphism of the constraint graph can be found into them.



no wine is sweet and strong ?

Positive constraints

A positive constraint is a structured as a rule with a condition part and the obligation part. A fact satisfies a positive constraint if every homomorphism from the condition part to the fact can be extended to a homomorphism of the obligation part to the fact. The example below expresses the fact that "a wine is necessarily associated with a winery". Positive constraints will be triggered each time a wine appears without a Winery attached to it with hasMaker relation.

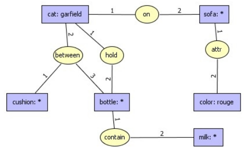


Facts

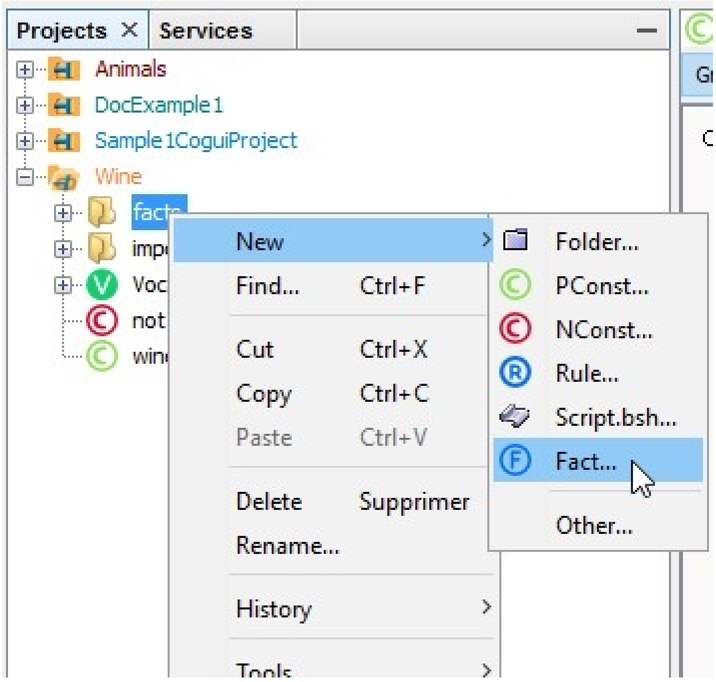
A Fact is labeled bipartite graph. One class of nodes (the concept nodes) is used to represent entities of the discourse universe. A concept node is labeled by a concept type (e.g. Painter, or a conjunctive type such as Painter,Catalan) and, possibly, by an individual (e.g. Picasso). A concept node which is labeled by a concept type without an individual is called a generic concept node. Such a node represents an unidentified element of the type. For instance, contrarily to a node labeled [Painter : Picasso] representing the painter Picasso, a concept node labeled [Painter] represents “a” painter.The second class of nodes represents the relationships between the entities (represented by the concept nodes). For instance, if Guernica is an individual representing the well-known painting realized by Picasso then a relation node labeled (hasPainted) could relate the concept node [Painter : Picasso] to the concept node [Painting : Guernica]. The edge between (hasPainted) and [Painter : Picasso] is labeled by 1, and the edge between (hasPainted) and [Painting : Guernica] is labeled by 2. This edge labeling is used to represent different roles (e.g. to distinguish the subject from the complement). It is also possible to say that two different concept nodes represent the same entity by linking them by a coreference link.

Picture below is described by a fact graph:

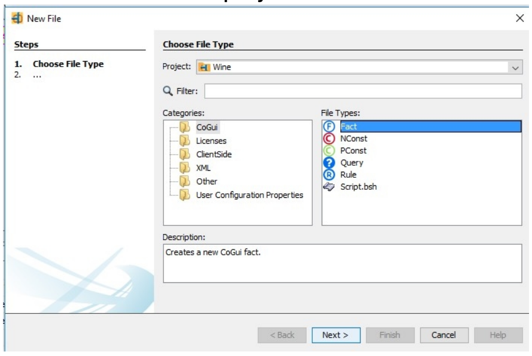
conceptual graph description of the picture:



Create a new fact:



If "Fact..." action does not already appears in menu choose "Other..." and select the type of CoGui object that you want to add in your project:



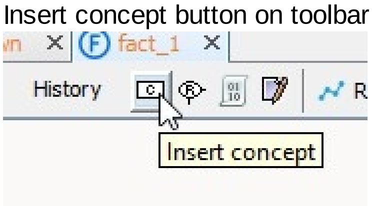
More about fact edition:

- Insert new concept
- Insert new relation
- Coreference
- Reduced edition

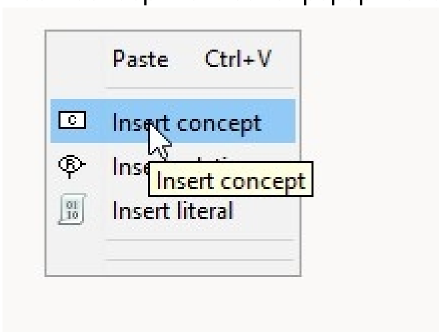
Insert new concept

There are many ways to create new concepts.

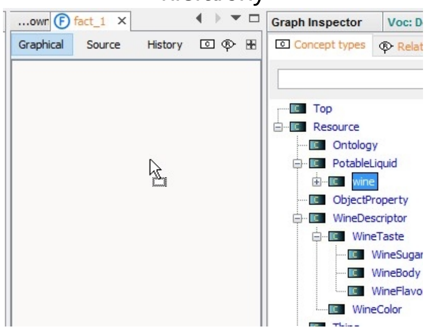
Insert concept button on toolbar



Insert concept from editor popup menu

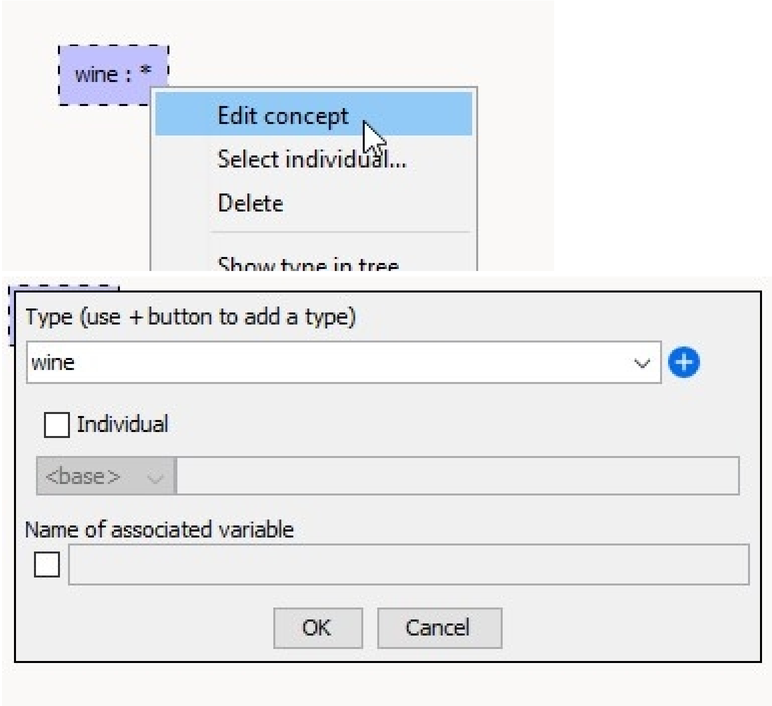


Drag a concept from the type hierarchy



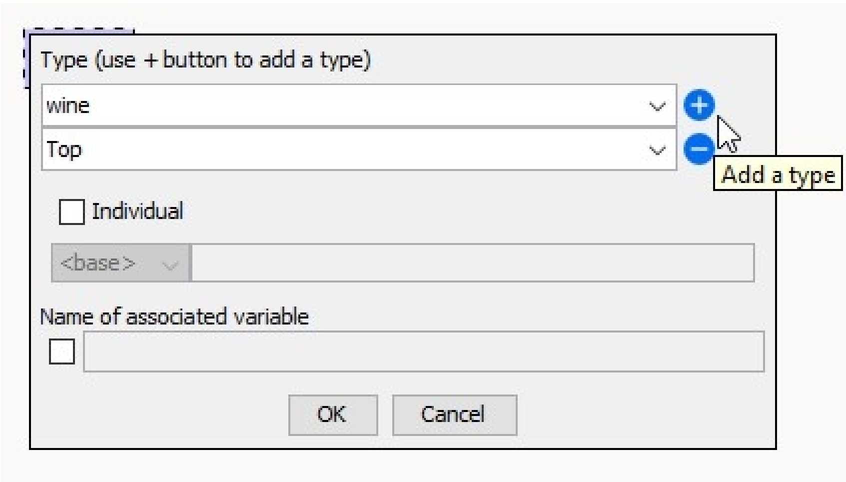
The third way (drag and drop) is the most effective since it informs in a single action the position and type of the new concept.
An alternative is to start by creating a relation and then complete it. The concepts will be created automatically with the type corresponding to the signature of the relation. See section Insert new relation.

Click twice on the concept vertex or use the popup menu to edit the newly created concept:



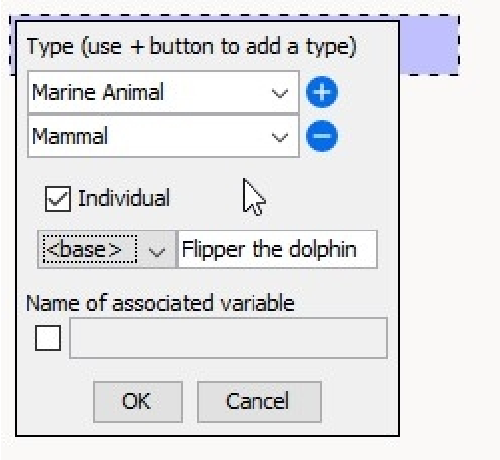
A concept can be associated to a conjunctive type.

Press  button to add a type field and select the second type:



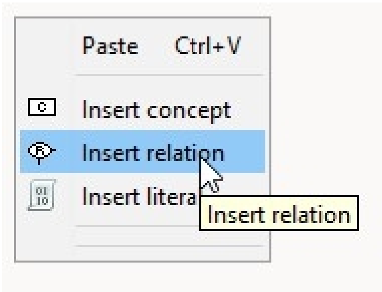
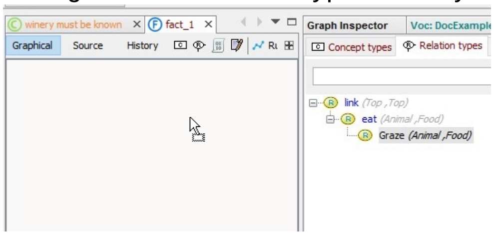
A concept with a conjunctive type:

An individual can also be associated to the concept. If this individual doesn't exist, it is automatically added to the project



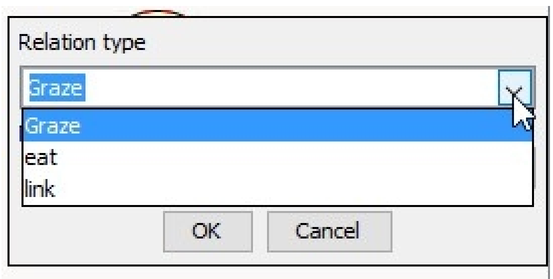
Insert new relation

There are many ways to create new relations.

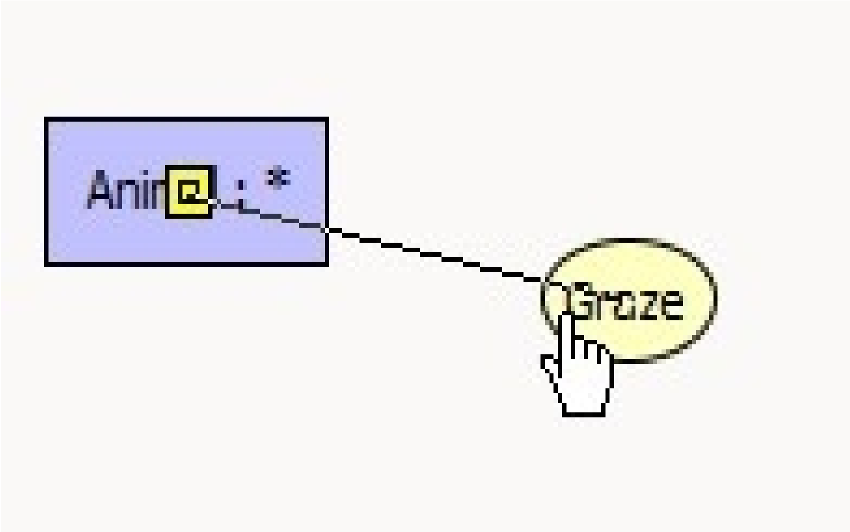
Insert concept button on toolbar	Insert concept from editor popup menu	Drag a relation from the type hierarchy
		

The third way (drag and drop) is the most effective since it informs in a single action the position and type of the new relation.

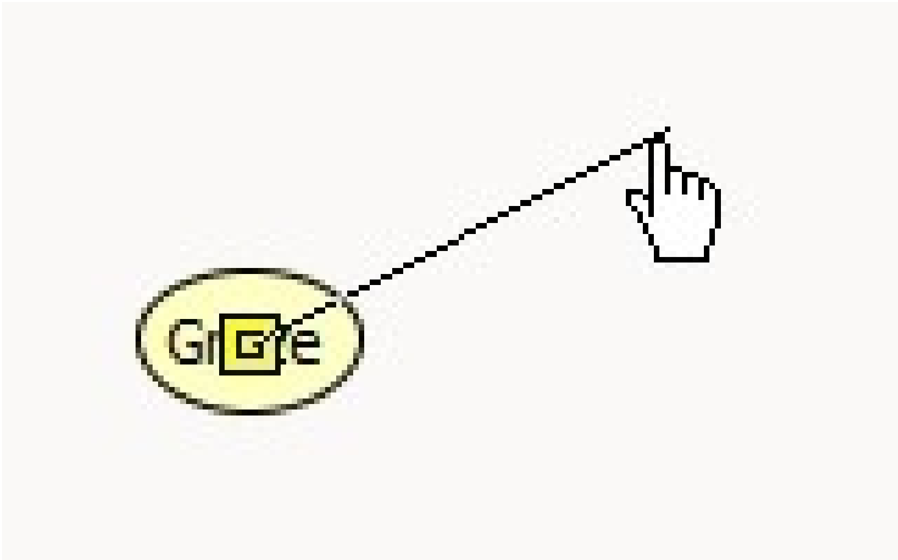
When the relation is created, double left click triggers a popup dialog to define type associated with selected relation:

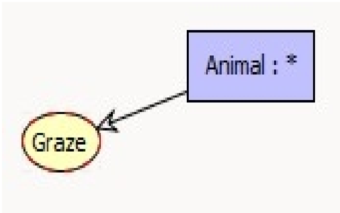


The insertion of the relationship is not sufficient, we must also link the relation to adjacent concepts by holding down the left mouse button and linking the yellow squares in the center of the vertices:

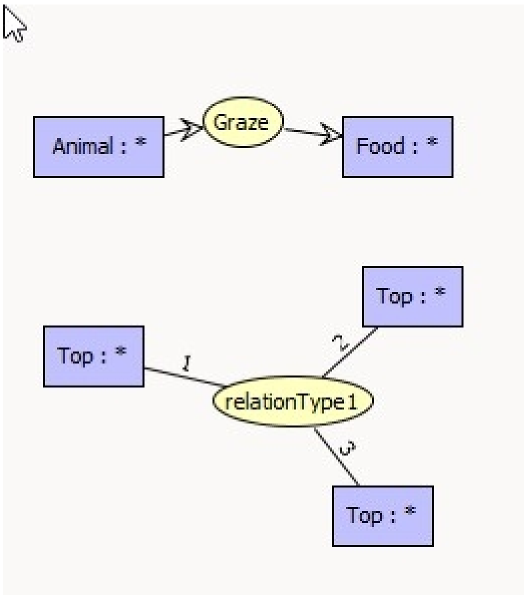


If the adjacent concepts are not already created you can quickly complete the relation by releasing the mouse button on the location of the future concept:

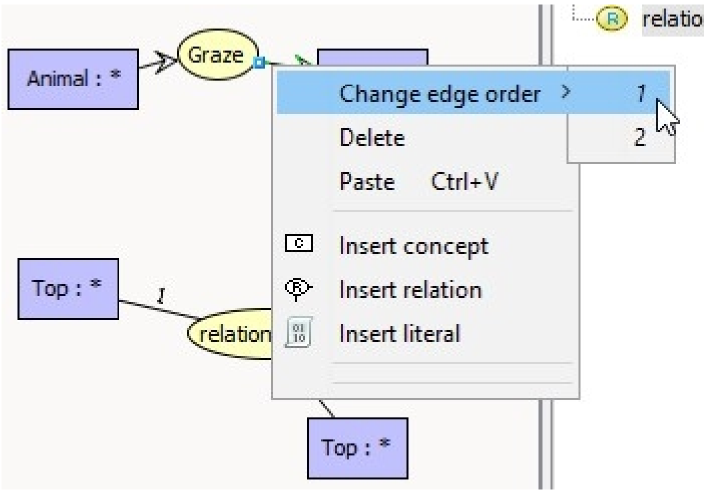




Because the relation signature is ordered. For binary relations the edges are ordered in a natural way by the subject has first parameter and the object has second parameter. For a greater arity edges between concepts and relation are associated with a number from 1 to relation arity.



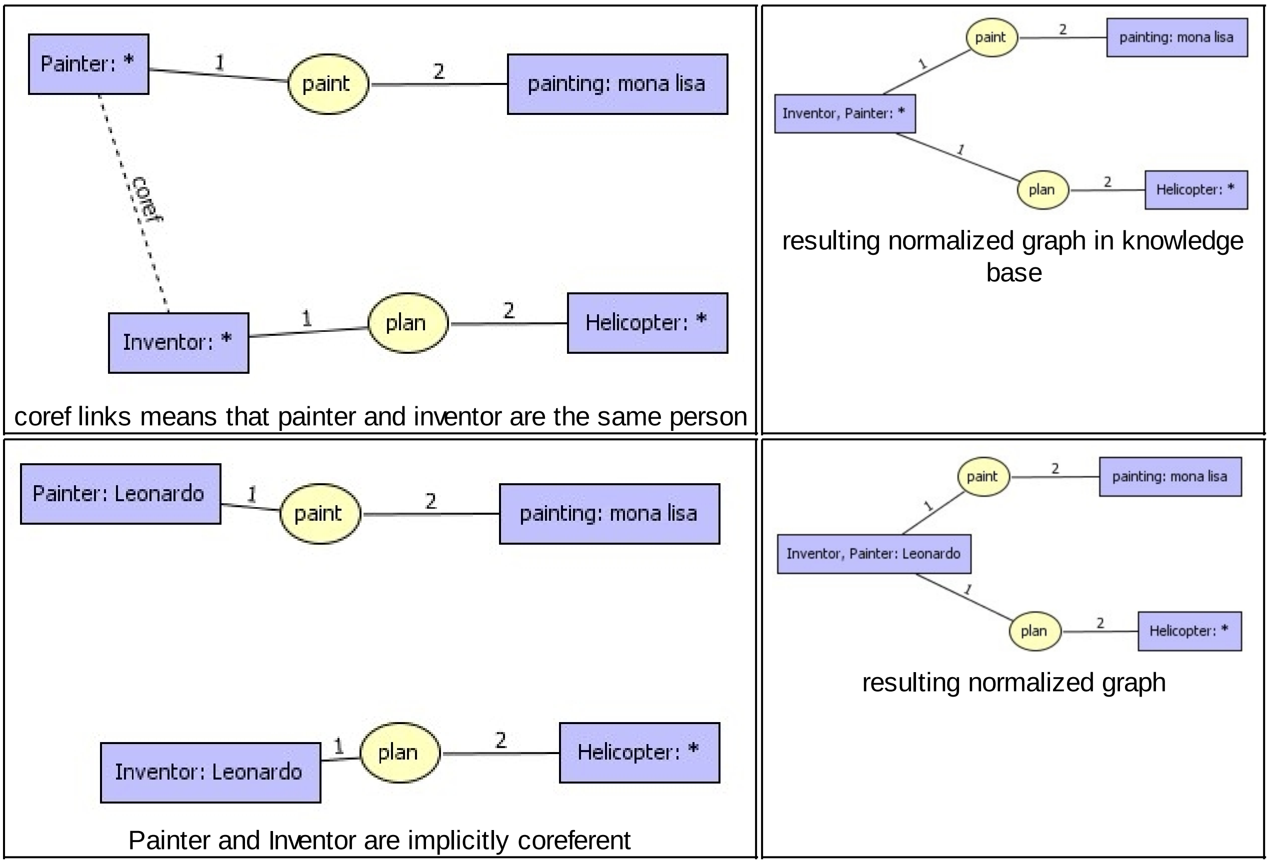
Edge order can be defined with popup menu on the selected edge:



You can also use double-click on selected edge, edge label successively takes all compatible values between the relation signature and the type of concept.
Of course, copy/paste mechanism can also be used to duplicate pieces of graphs inside the graph or from one to another. Pending edges cannot be copied.

Coreference

It is also possible to say that two different concept nodes represent the same entity by linking them by a coreference link.



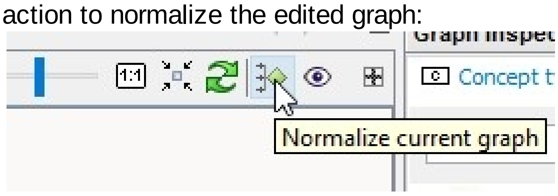
coref links means that painter and inventor are the same person

resulting normalized graph in knowledge base

Painter and Inventor are implicitly coreferent

resulting normalized graph

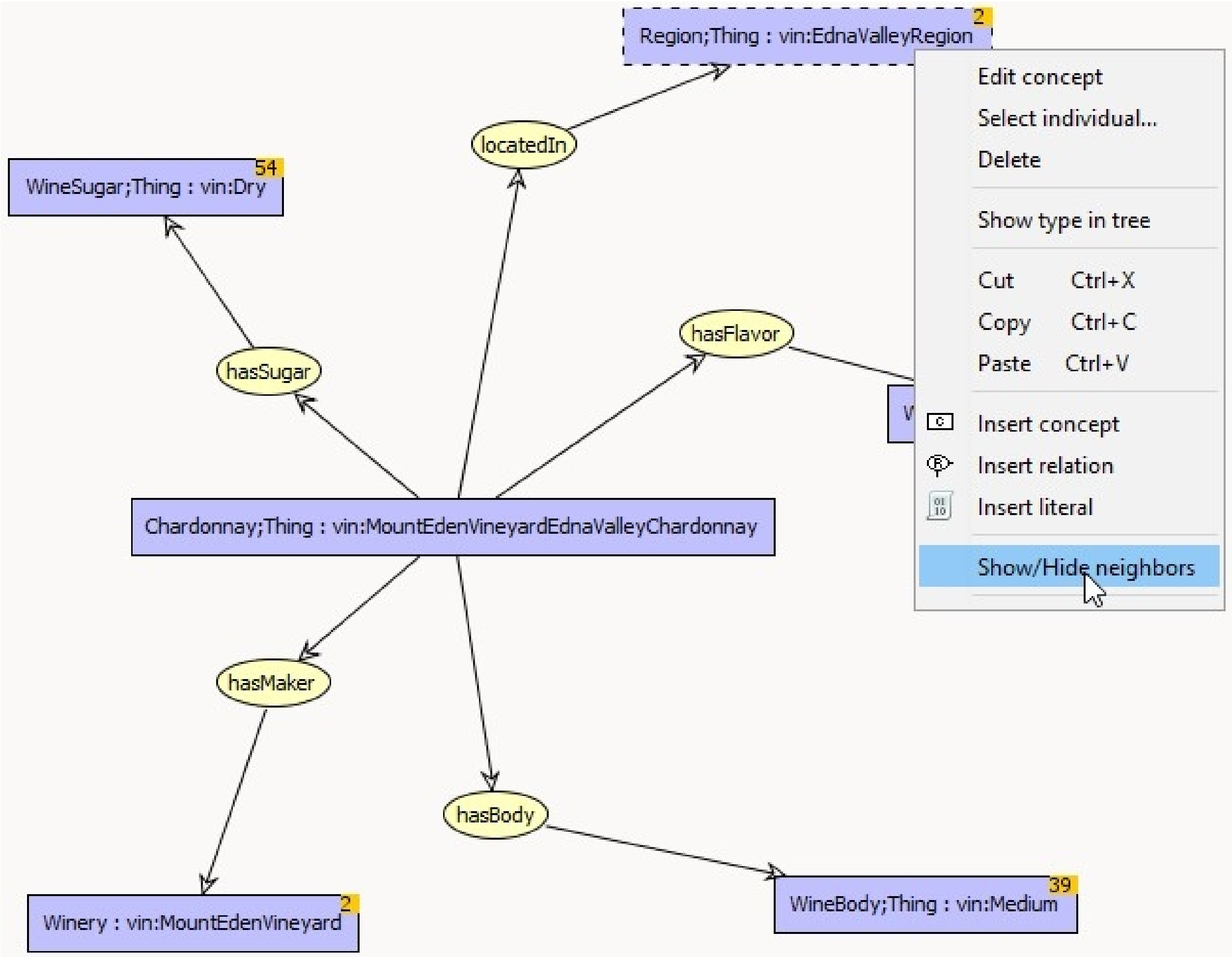
The editor toolbar propose an action to normalize the edited graph:



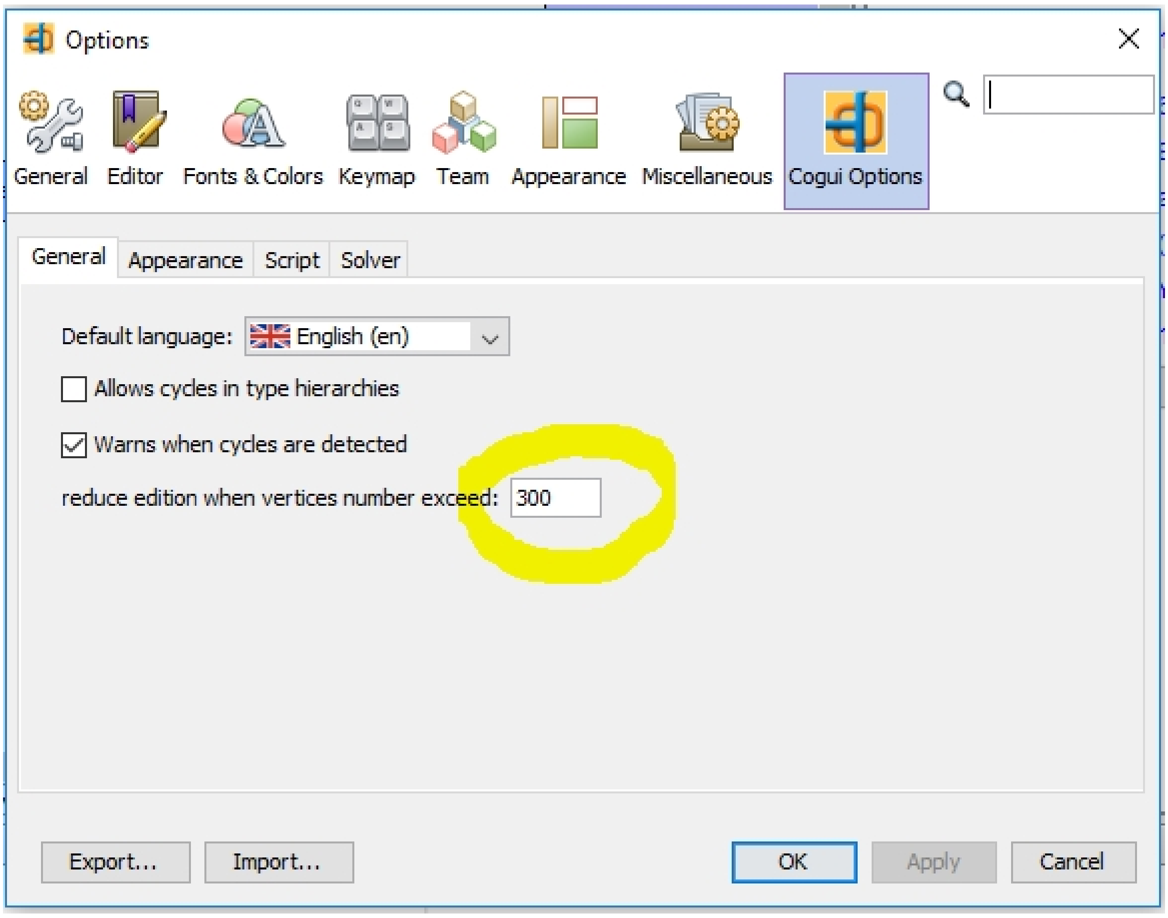
See section Sum, split and normalization for more about the normalization of the graphs

Reduced edition

If a graph is too big it could be very tedious (or impossible) to edit it entirely. In this case, CoGui provide a reduced edition mode. Just a part of the graph is displayed. Visible relations are always completed, but displayed concept can have hidden neighbors. Example below shows concepts with hidden neighbors, the number of hidden neighbors is displayed on the upper right corner of the concept.



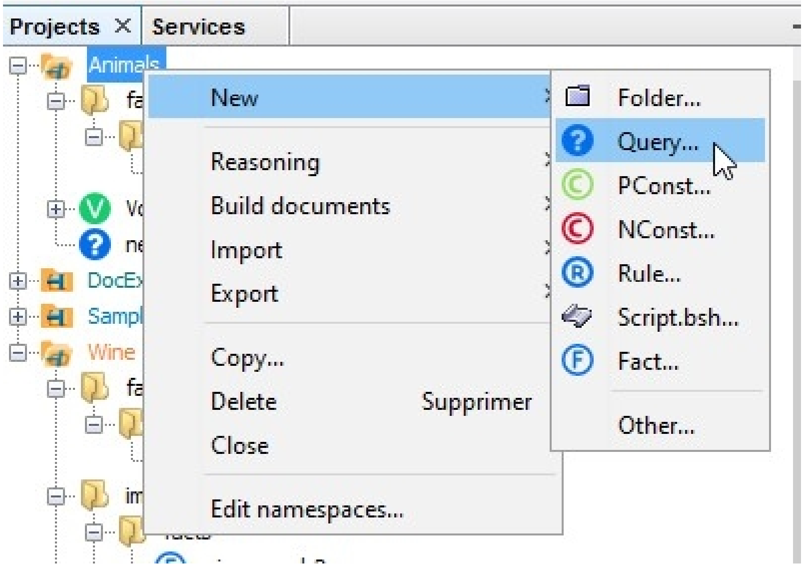
The limits to choose partial edition rather than whole graph edition can be changed in CoGui options:



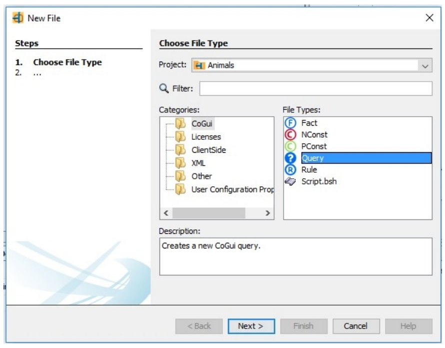
If a big graph is directly edited from the project tree, some vertices are chosen by default. The user generally prefers to display and update a precise piece of graph. To correct errors inside the graph, choose the edit option on the message in the error view and the graph will be automatically opened with the vertices that generated the error. When another error is selected, the previous edition is replaced by the edition of the new concerned vertices.
To edit parts of the graph without any error, we must query the graph (See section Querying) and browse through projections in the result view.

Queries

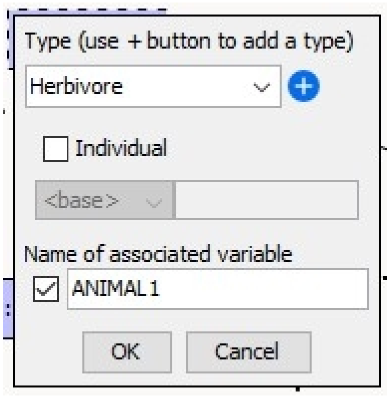
Create a new query:



If "Query..." action does not already appears in menu choose "Other..." and select the type of CoGui object that you want to add in your project:



Editing queries is very similar to editing Facts. In addition some variables can be added. Read section Querying for more about the use of variables.



Reasoning

This section groups the following operations on graphs:

Inspecting facts
- Graph measurement, redundancy
- Classification
- Check consistency

Applying rules
Sum, split and normalization
Querying

Inspecting facts

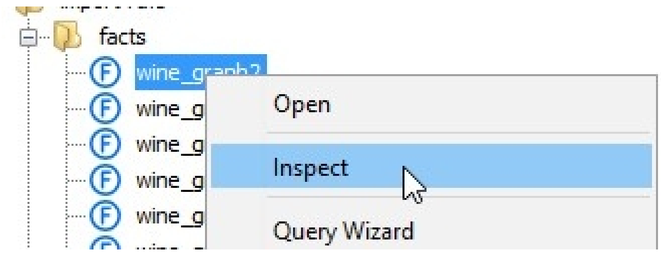
CoGui provide several tools to compute graph properties and compare them:

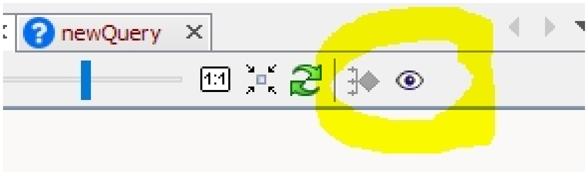
- Graph measurement, redundancy
- Classification
- Check consistency

Graph measurement, redundancy

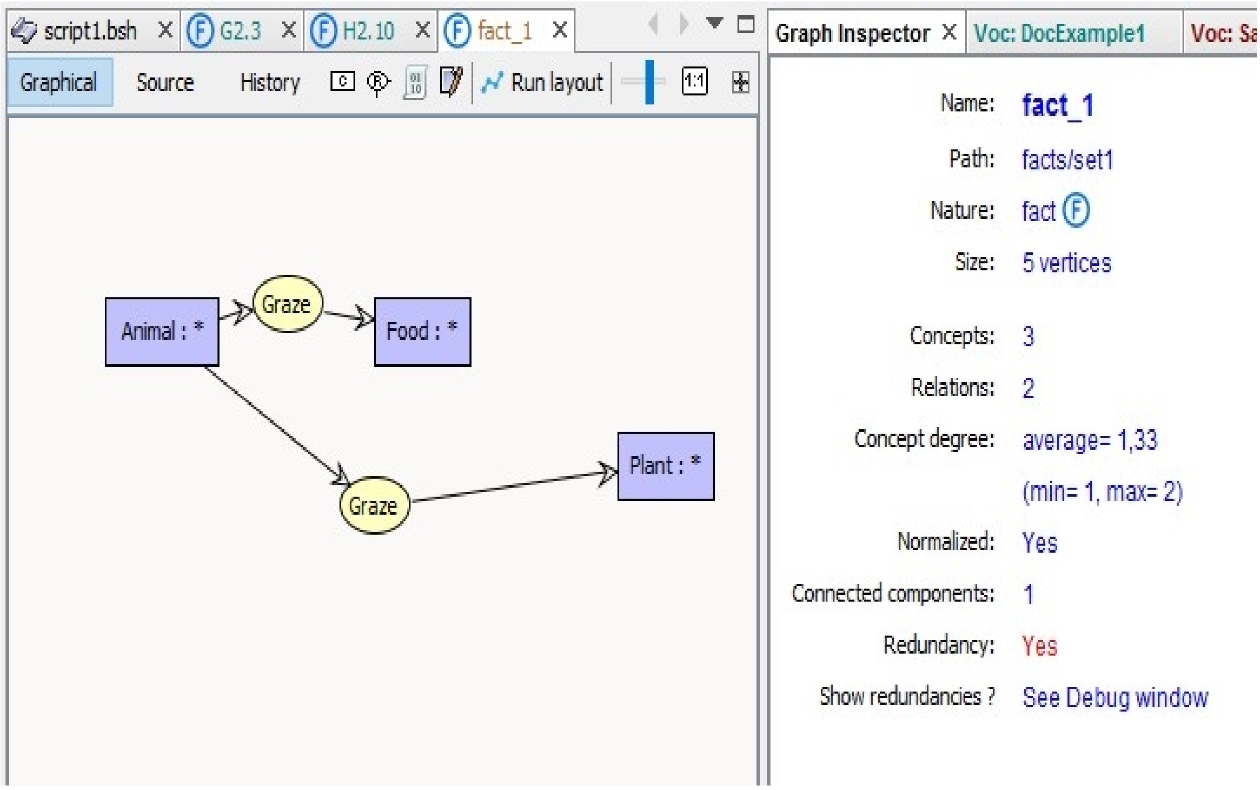
Each kind of graph can be inspected to know:

- the number of connected components
- the size of the graph individuals/concepts/relations
- the degree of the vertices min, max and average
- is it normalized ?
- is it redundant ?





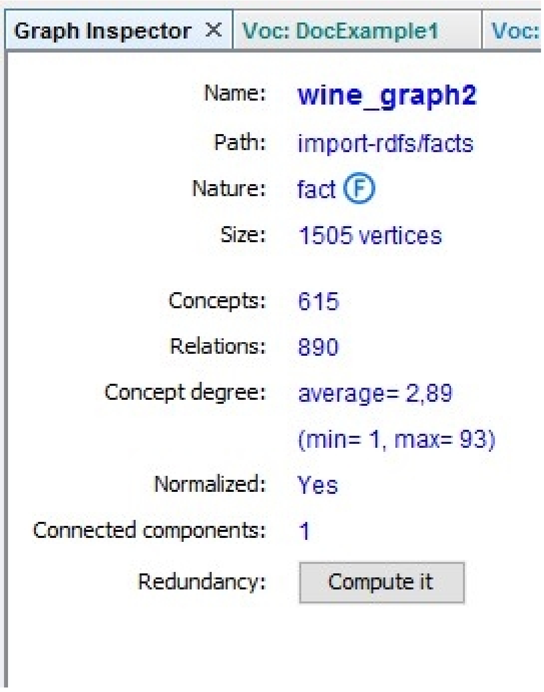
A View is opened named Graph Inspector



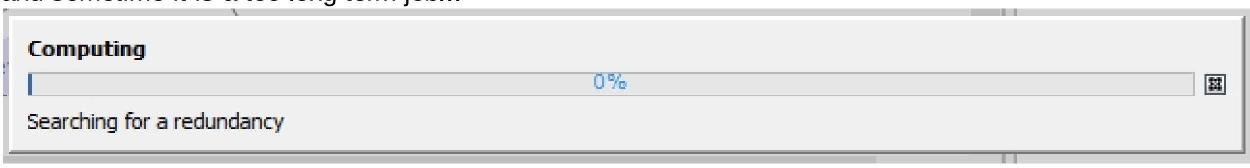
Redundancy

A redundant graph can trigger a positive constraint violation when irredundant form will not.
Detecting and computing redundancies can be an heavy operation, the wizard doesn't automatically provide it.
Click on button to run algorithm.

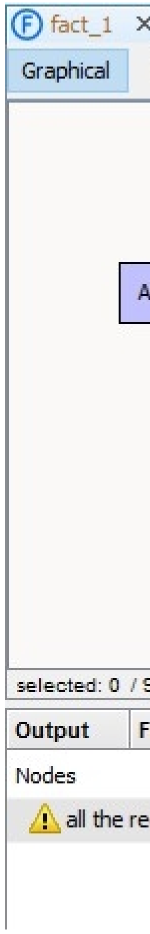
When the graph is too big, redundancy is not automatically computed. Click on button to compute it:



and sometime it is a too long term job...

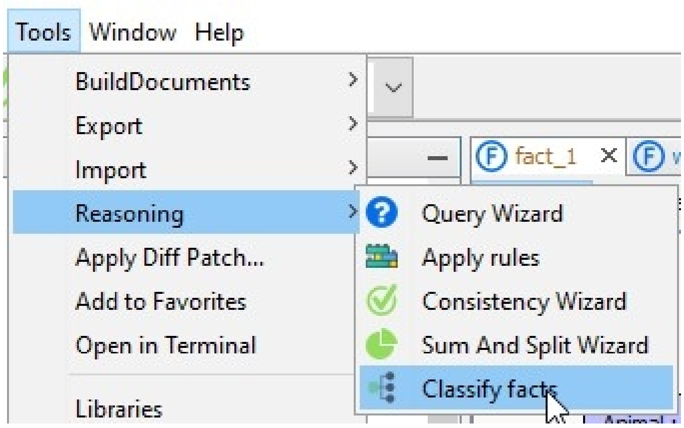


If redundanc

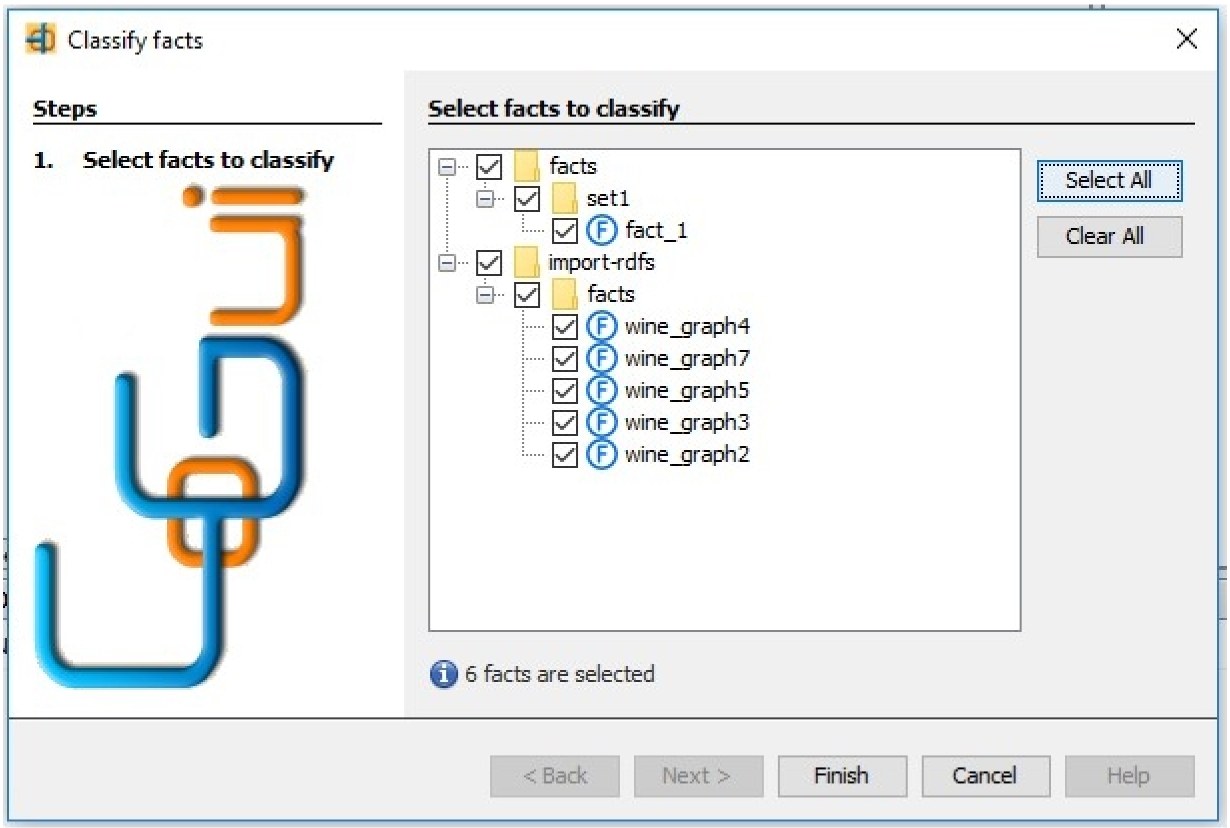


Classification

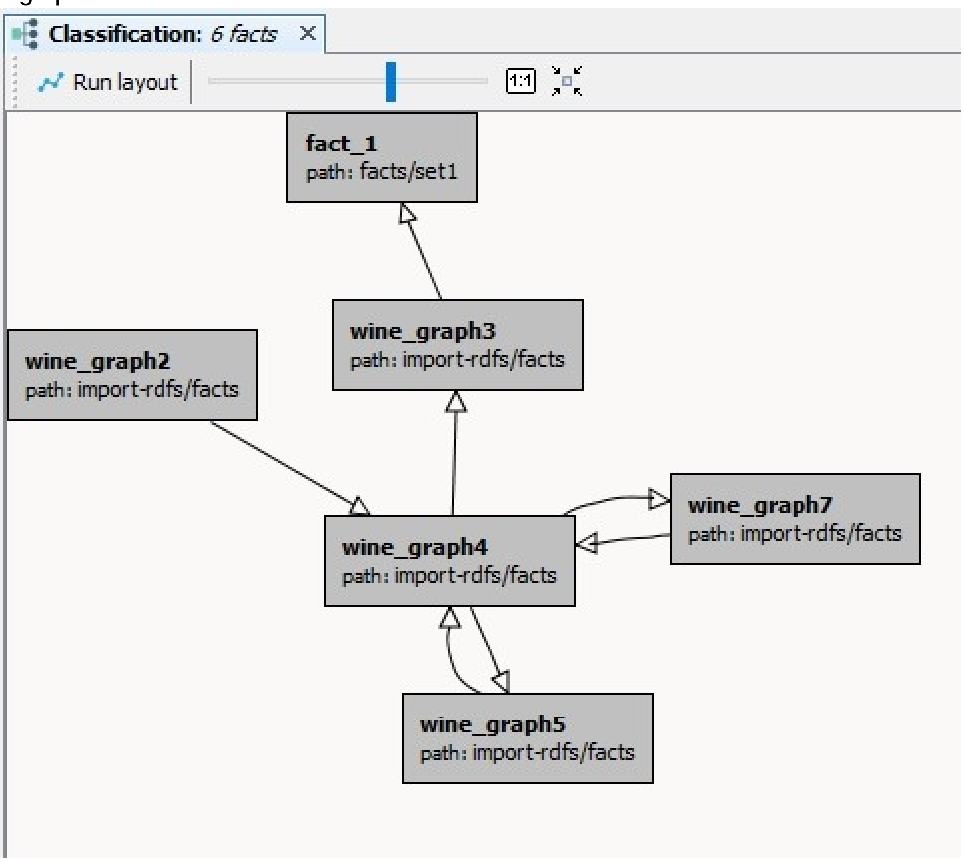
Facts can be analyzed individually, they can also be compared This is the purpose of the classification wizard.
Tools/Reasoning sub-menu provide a wizard to compute a classification between facts of the project:



Select facts to compare:



Classification graph viewer:



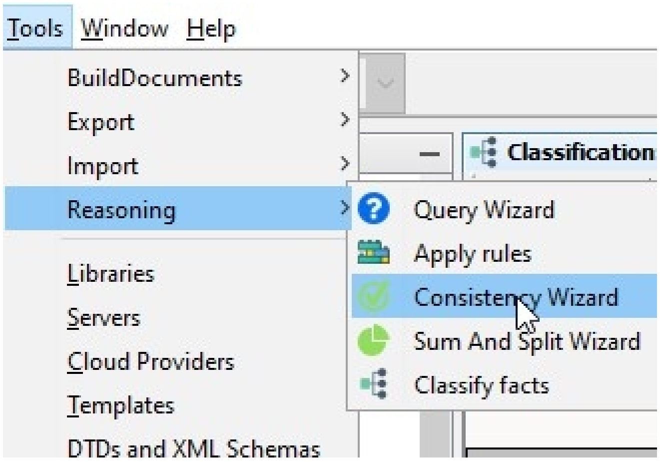
CoGui builds a graph with where vertices represent fact graphs and arcs represent the subsumption relations.
example interpretation:

- graph shows that wine_graph2 have a least one projection into wine_graph4.
- graph fact_1 is an empty graph that why every graphs subsume it.

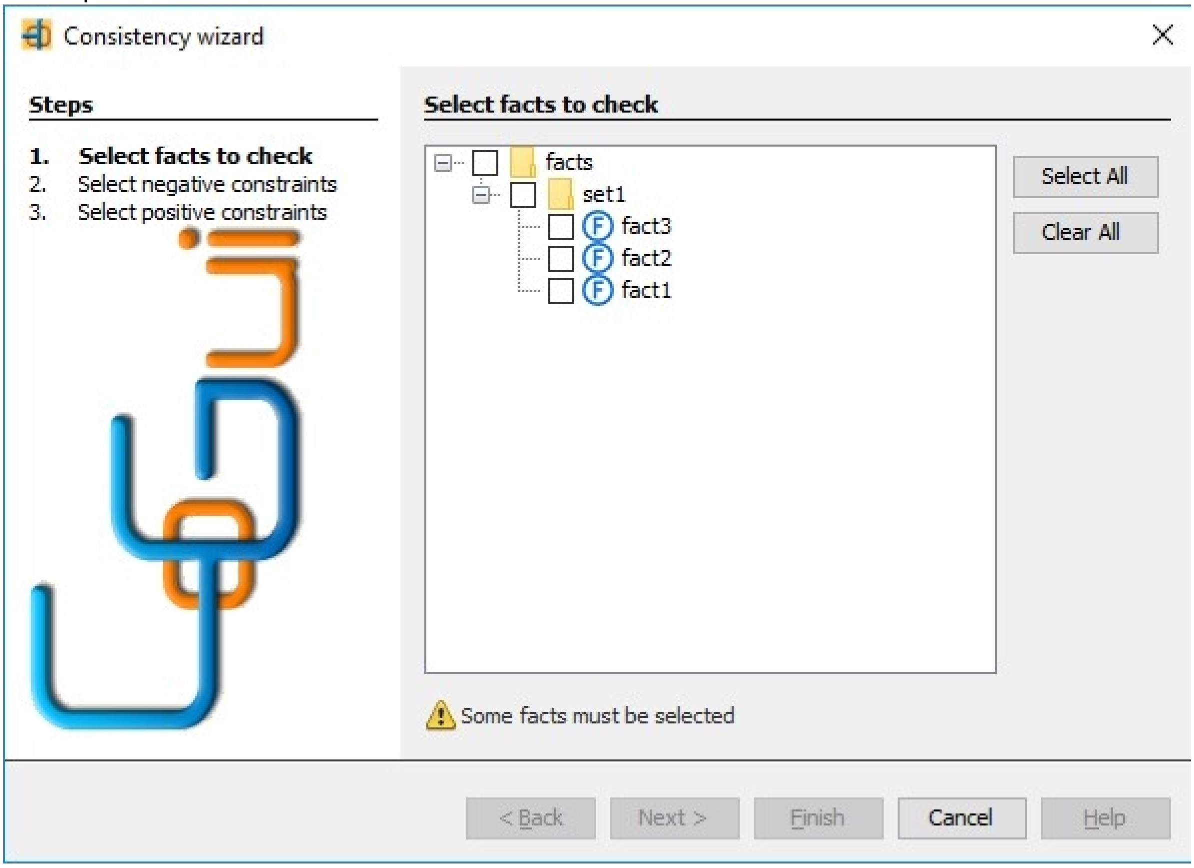
- wine_graph4, wine_graph5 and wine_graph7 belongs to the same strongly connected component, it means that they are equivalent (but not equals if they are redundant).

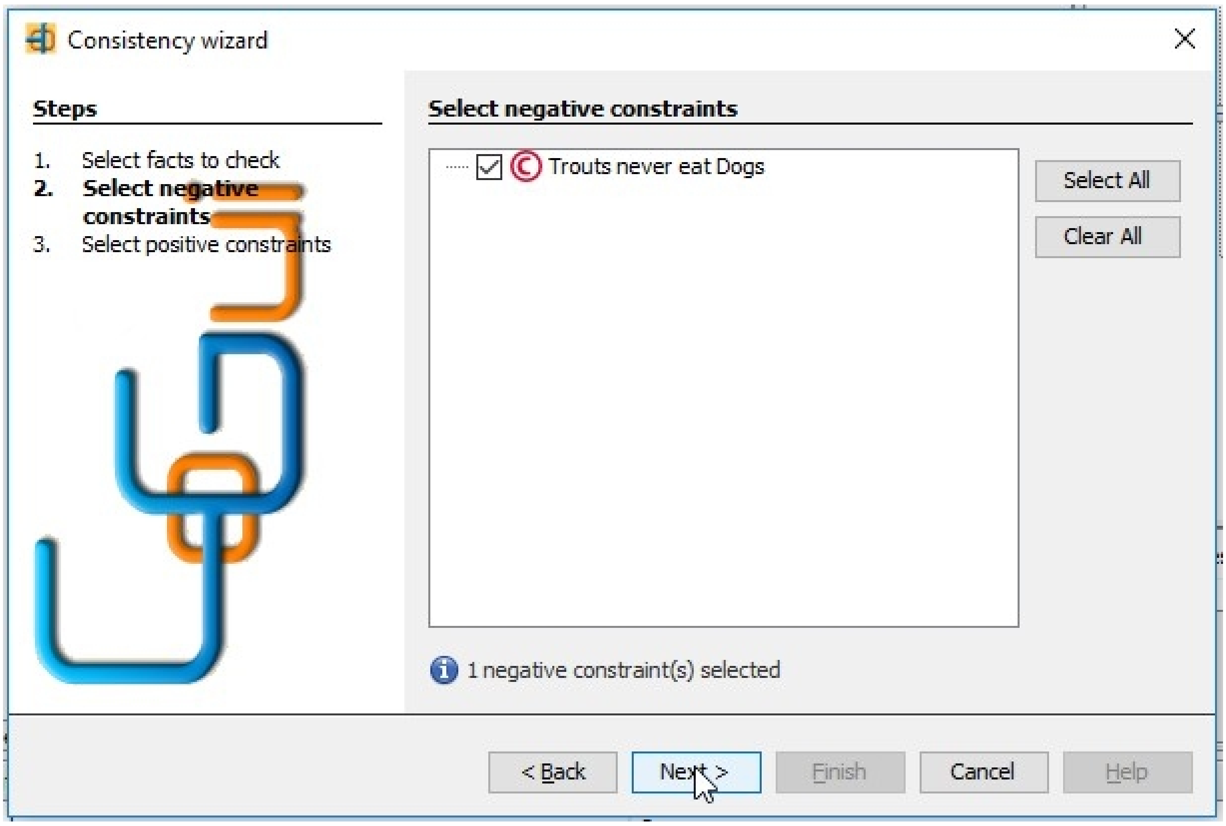
Check consistency

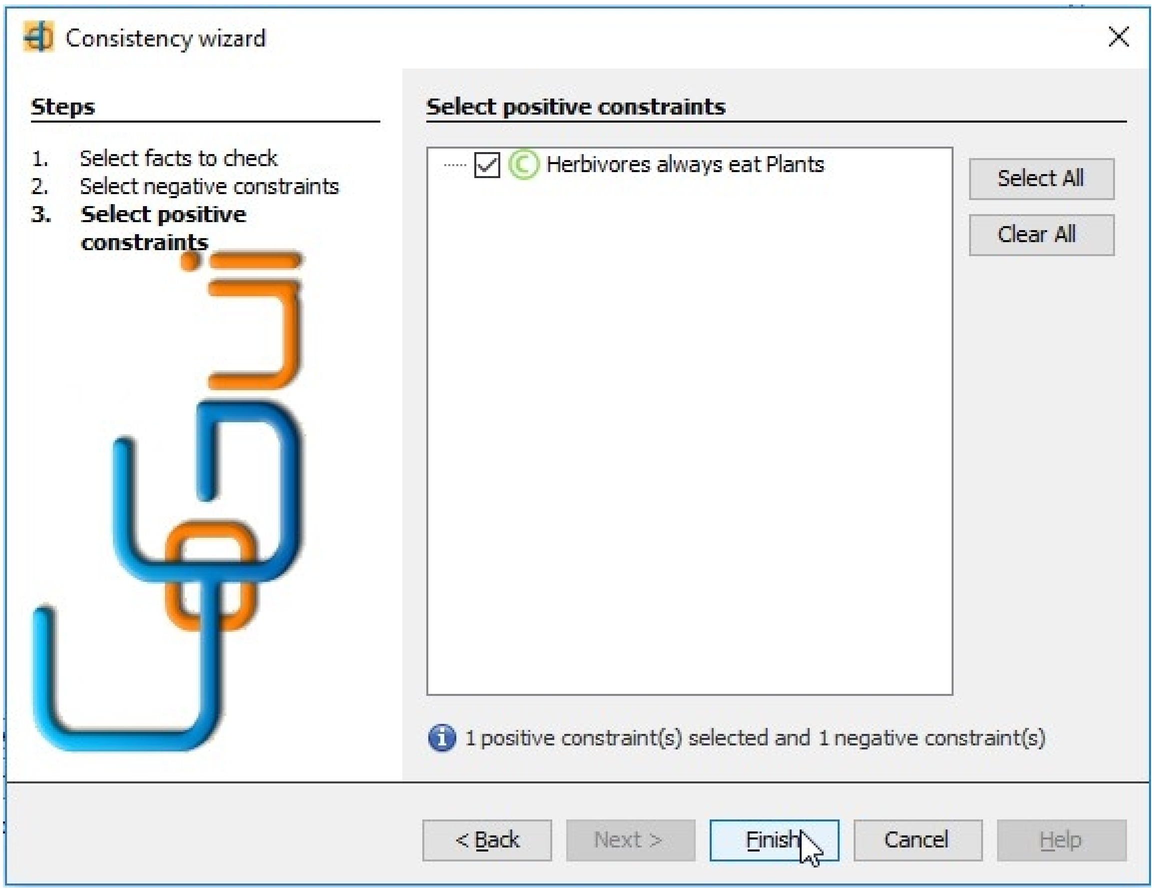
Constraints can be defined to control the set of facts. See more about Constraints
The Toolbar and Tools/Reasoning sub-menu provide a wizard to check consistencies.



The 3 steps of the wizard:

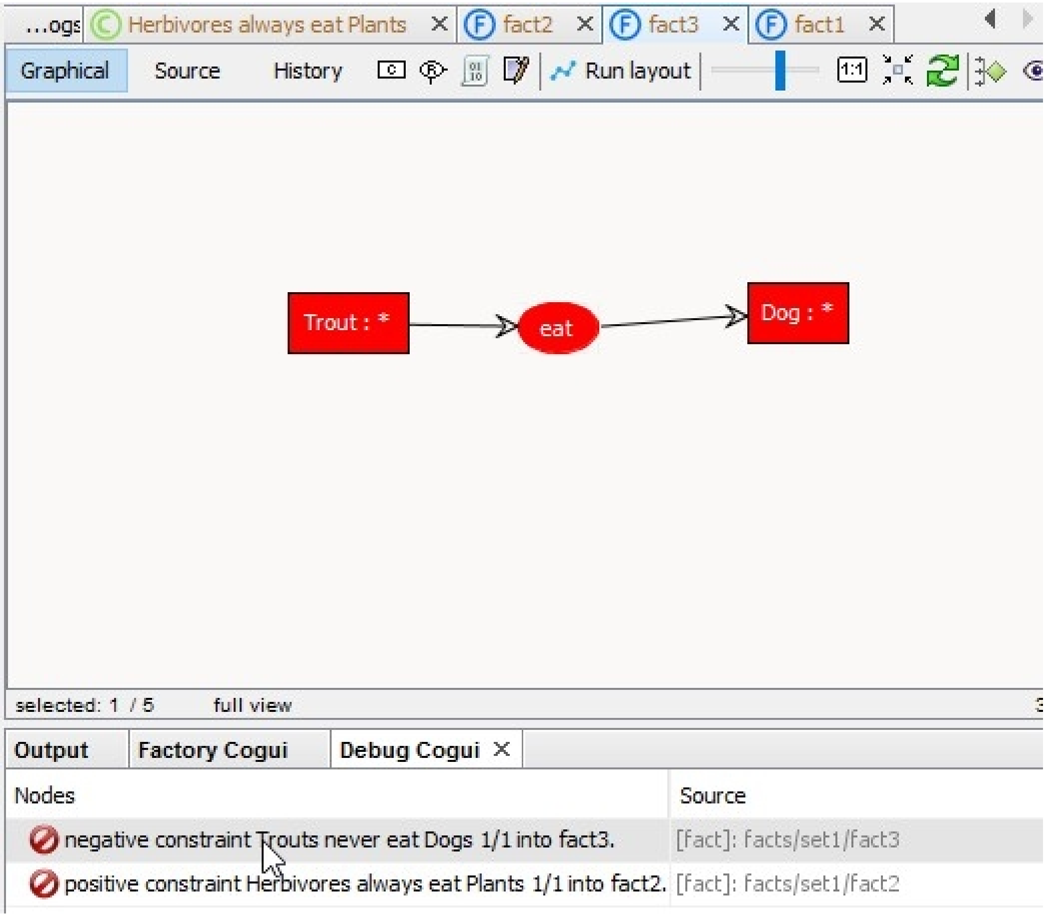




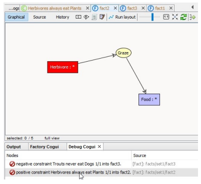


The wizard triggers messages in the Debug window when some constraints are violated.

Double click on messages to reach the part of the graph containing inconsistencies:



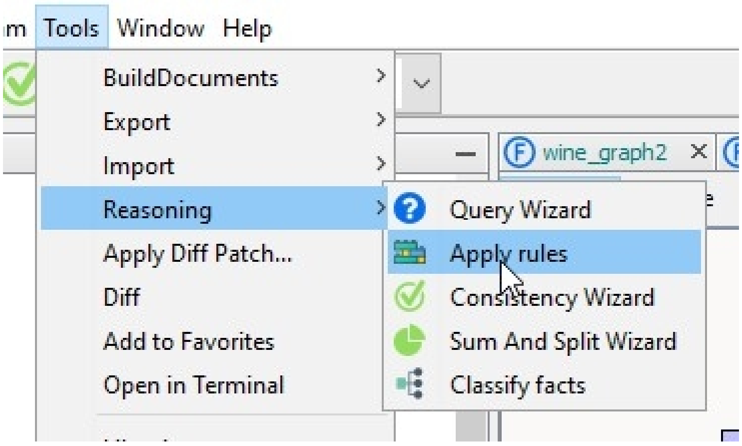
a negative constraint is violated

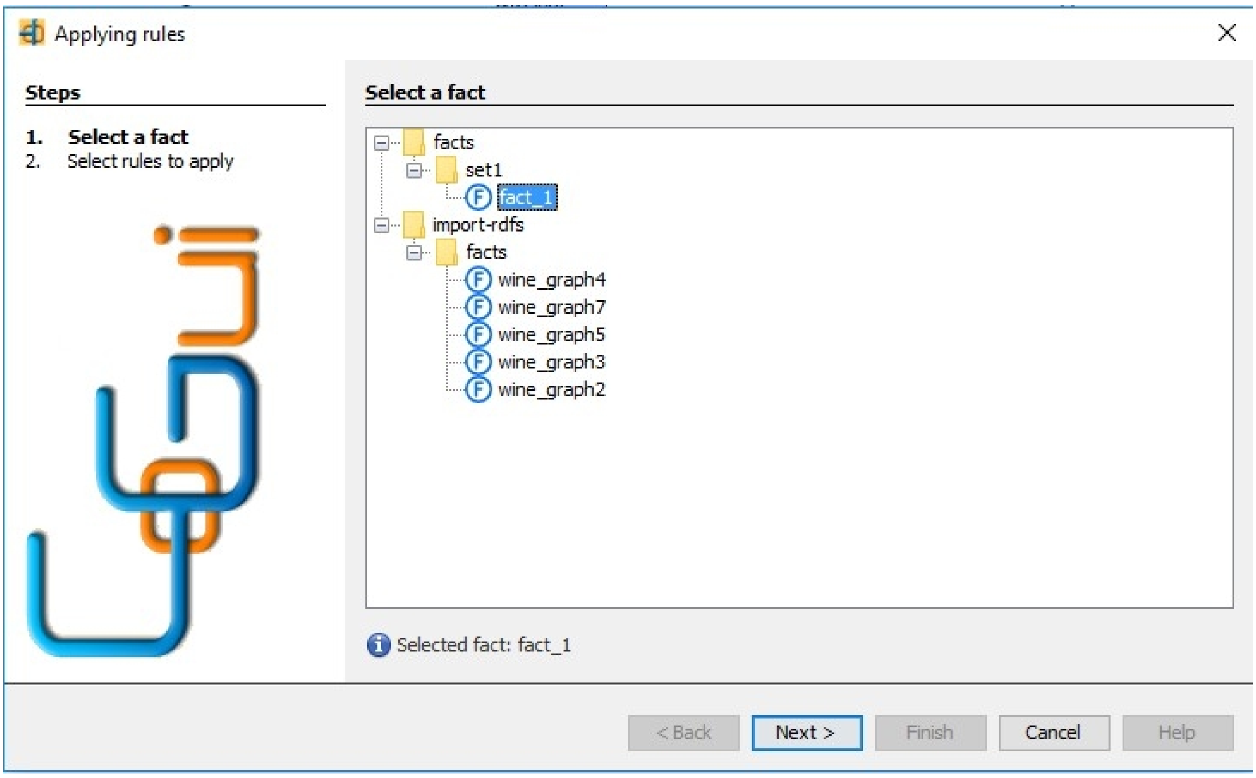


a positive constraint is violated

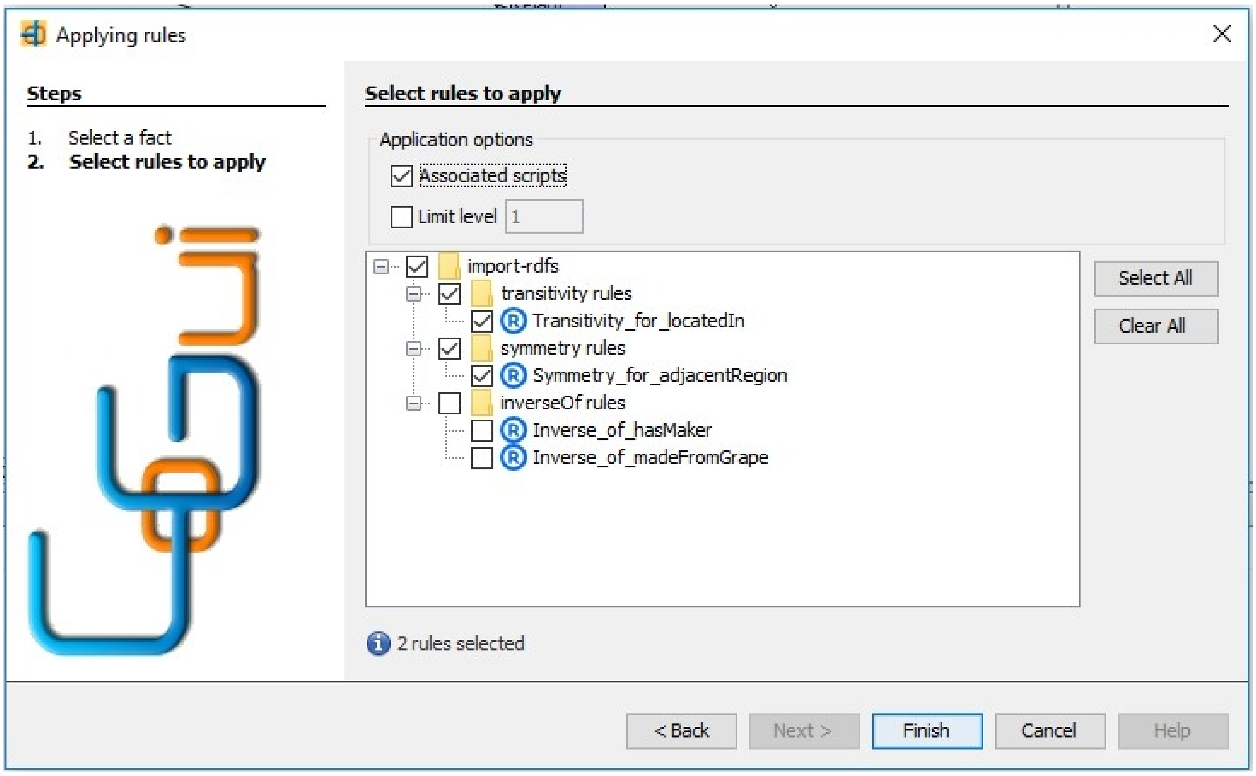
Applying rules

For now cogui only implements forward chaining technique and applies rules to the facts in order to produce new facts. The rule assistant helps to apply a selected set of rules to a graph. This operation can be launched for saturation or followed and visualized step by step. The new resulting graph can be added to the knowledge base.



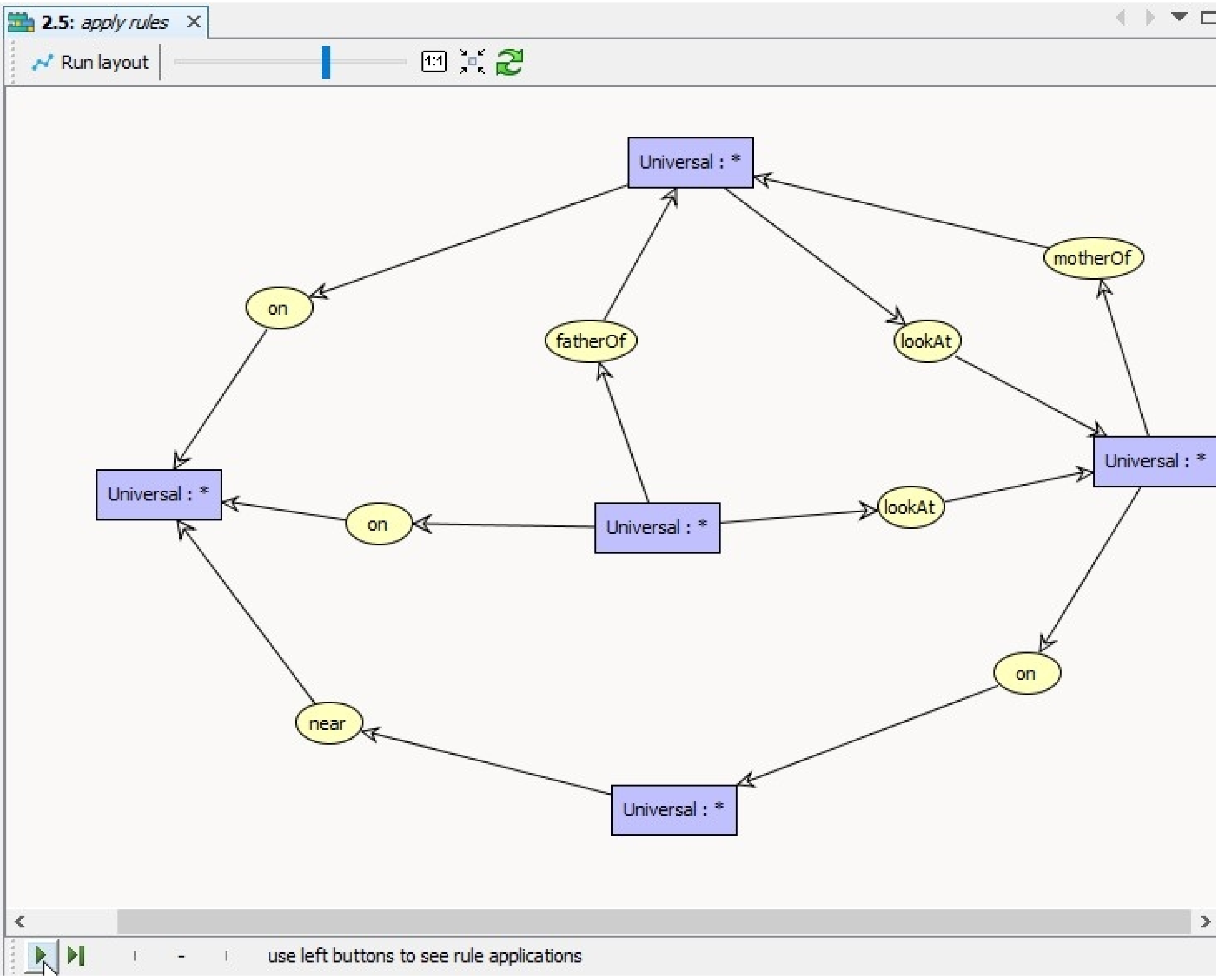


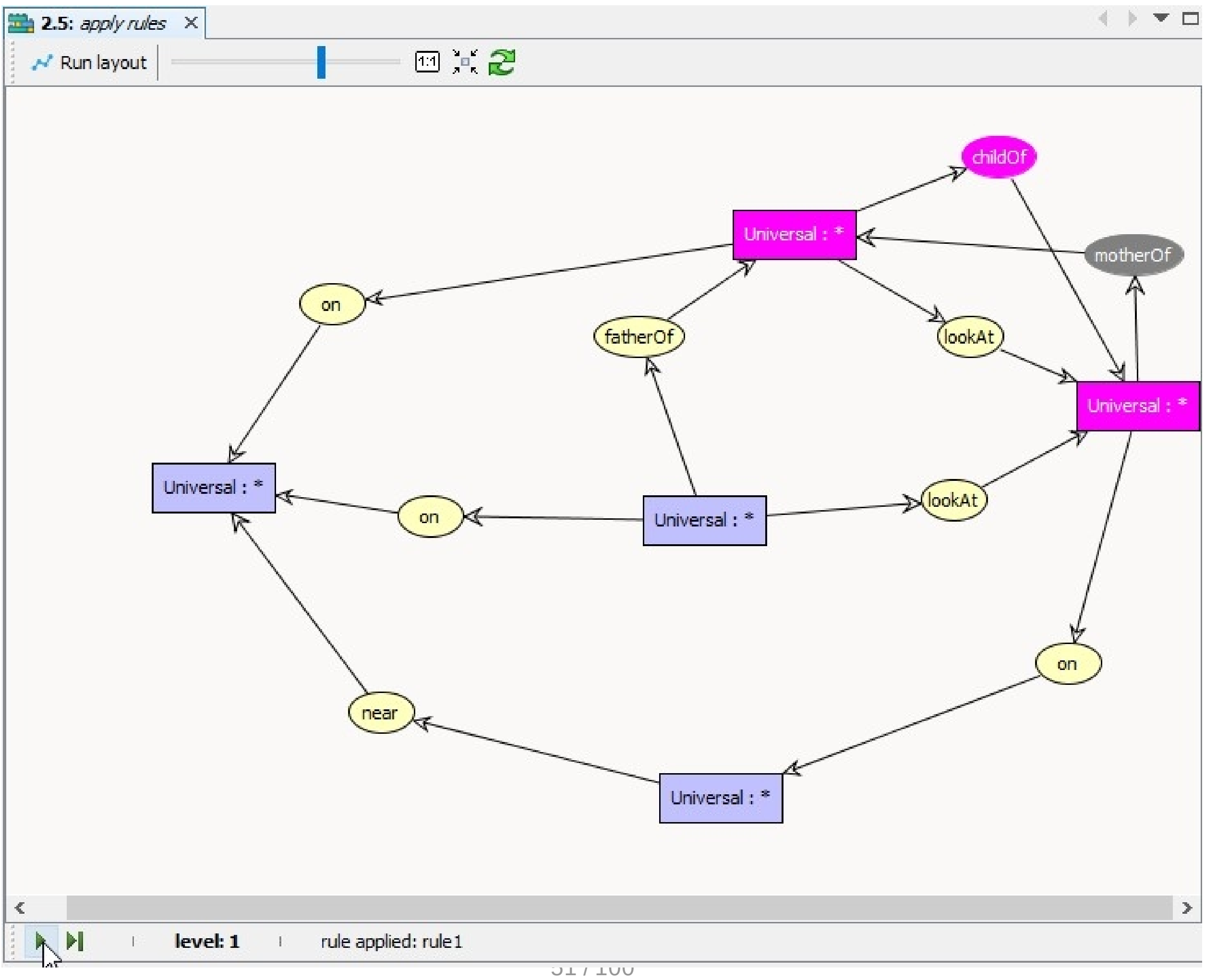
Select a unique graph



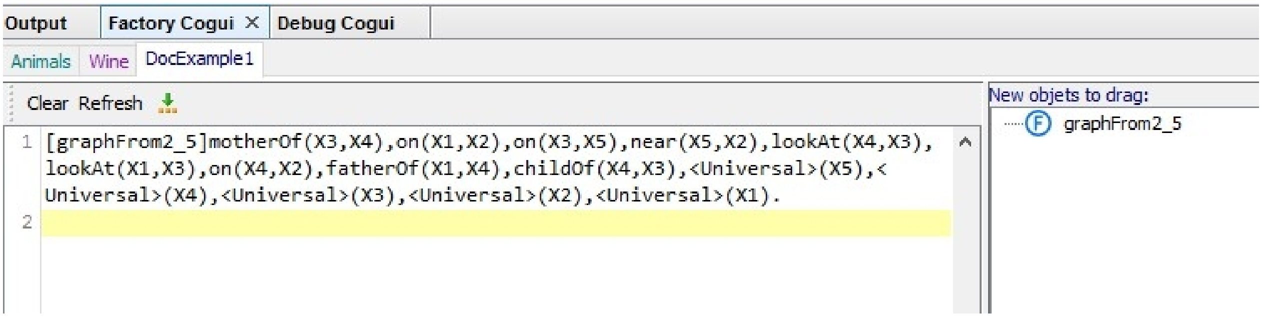
Select rules to apply

The wizard opens a windows to command and visualize rules application step by step:


2.5: apply rules
Run layout
Universal : *
motherOf
on
fatherOf
lookAt
Universal : *
Universal : *
on
Universal : *
lookAt
on
near
Universal : *
use left buttons to see rule applications

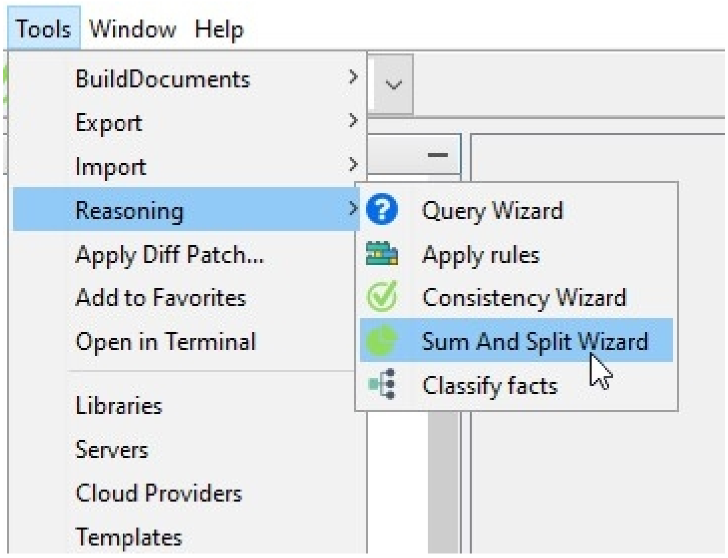

2.5: apply rules
Run layout
childOf
Universal : *
motherOf
on
fatherOf
lookAt
Universal : *
Universal : *
on
Universal : *
lookAt
on
near
Universal : *
level: 1
rule applied: rule1

If you want to save the resulting graph at any step of the saturation process, you can use the Datalog+ Factory (see section To/From Datalog+)



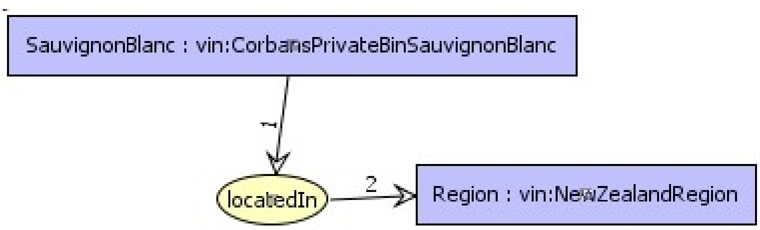
Sum, split and normalization

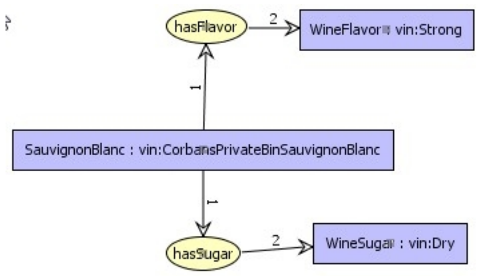
Tools/Reasoning sub-menu provide a wizard to sum and normalize facts of the project:



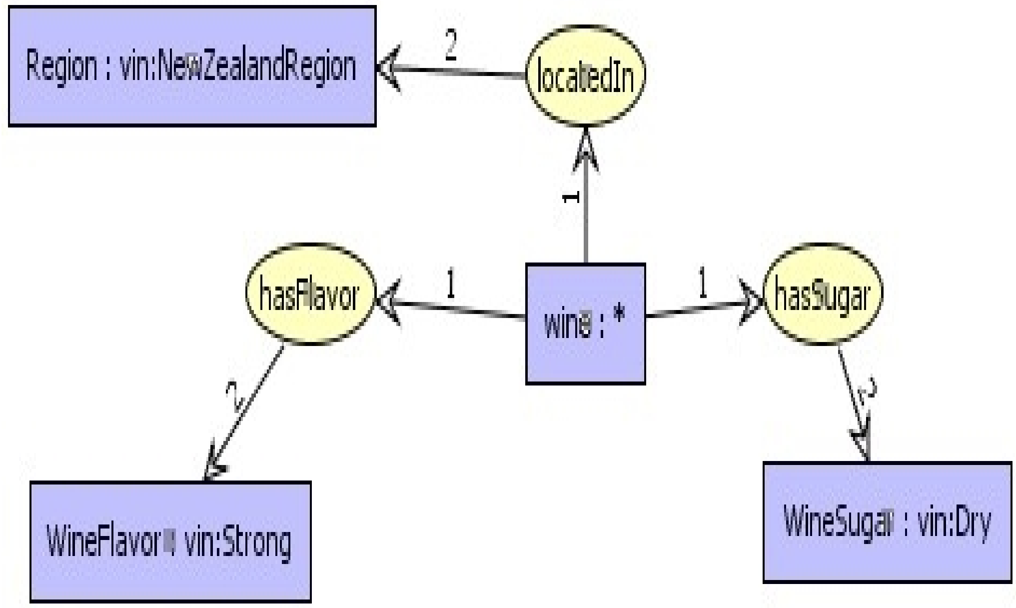
Sum

Knowledge base is organized by sets of facts.Sometime it can be interesting to sum these graphs. Example below presents two graphs first contains geographical information,second contains information about flavor and sweetness.

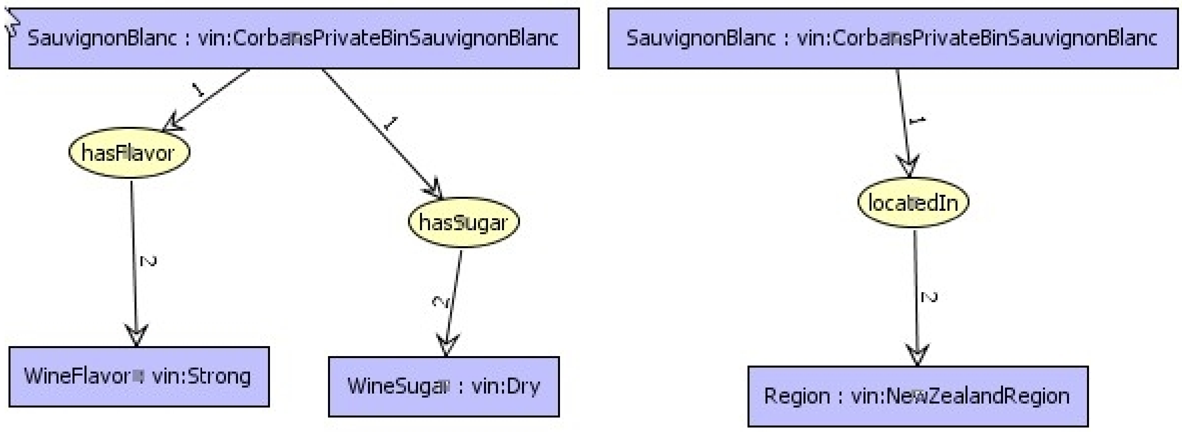




neither graph1 nor graph2 match with this query:



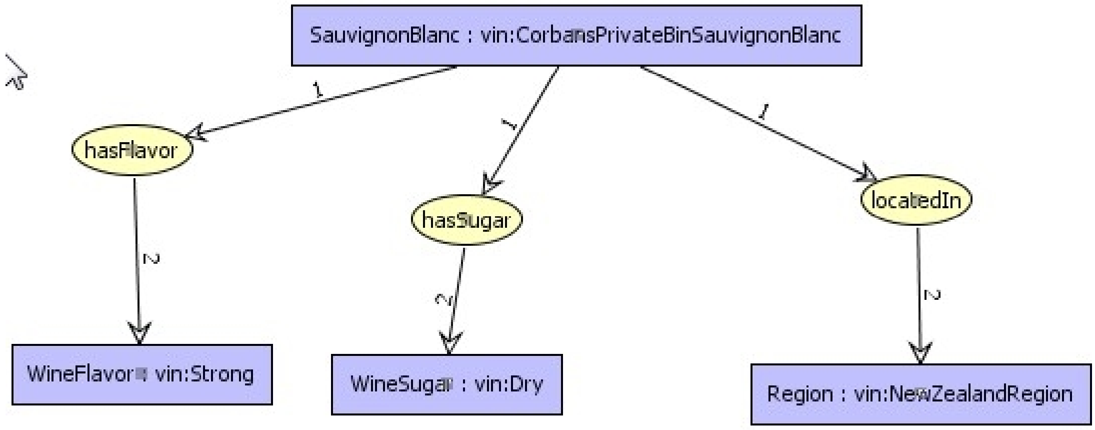
CoGui provides a wizard to sum graphs. It consist in merging graphs into the same sum graph.
the resulting graph is shown below :



not normalized sum of graph1 and graph2

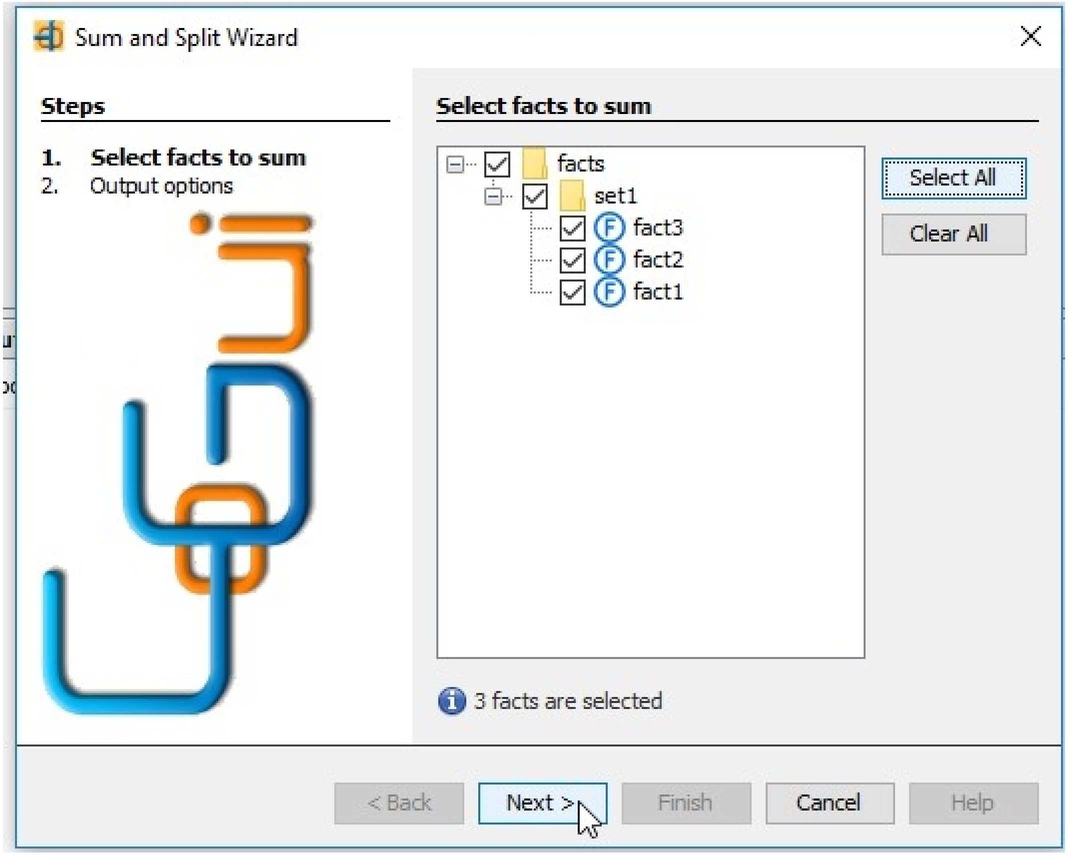
Normalize

We obtain a sum graph which is not a normal simple graph. A normal SG is such that each concept is uniquely coreferent with itself. Note that an implicit coreference link exists between the two nodes representing the individual vin:CorbansPrivateBinSauvignonBlanc. Normalization performs the fusion of nodes associated to the same individual. See below the normalized sum of graph1 and graph2. Resulting graph give an answer to query above.

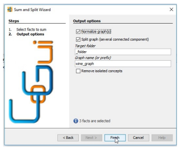


normalized sum of graph1 and graph2

Sum and Split wizard



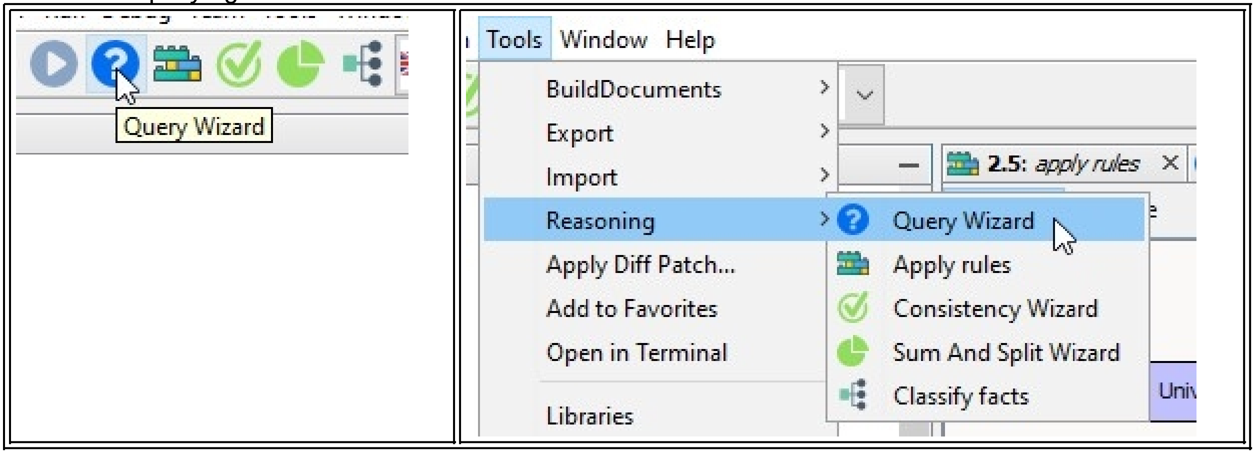
Select facts



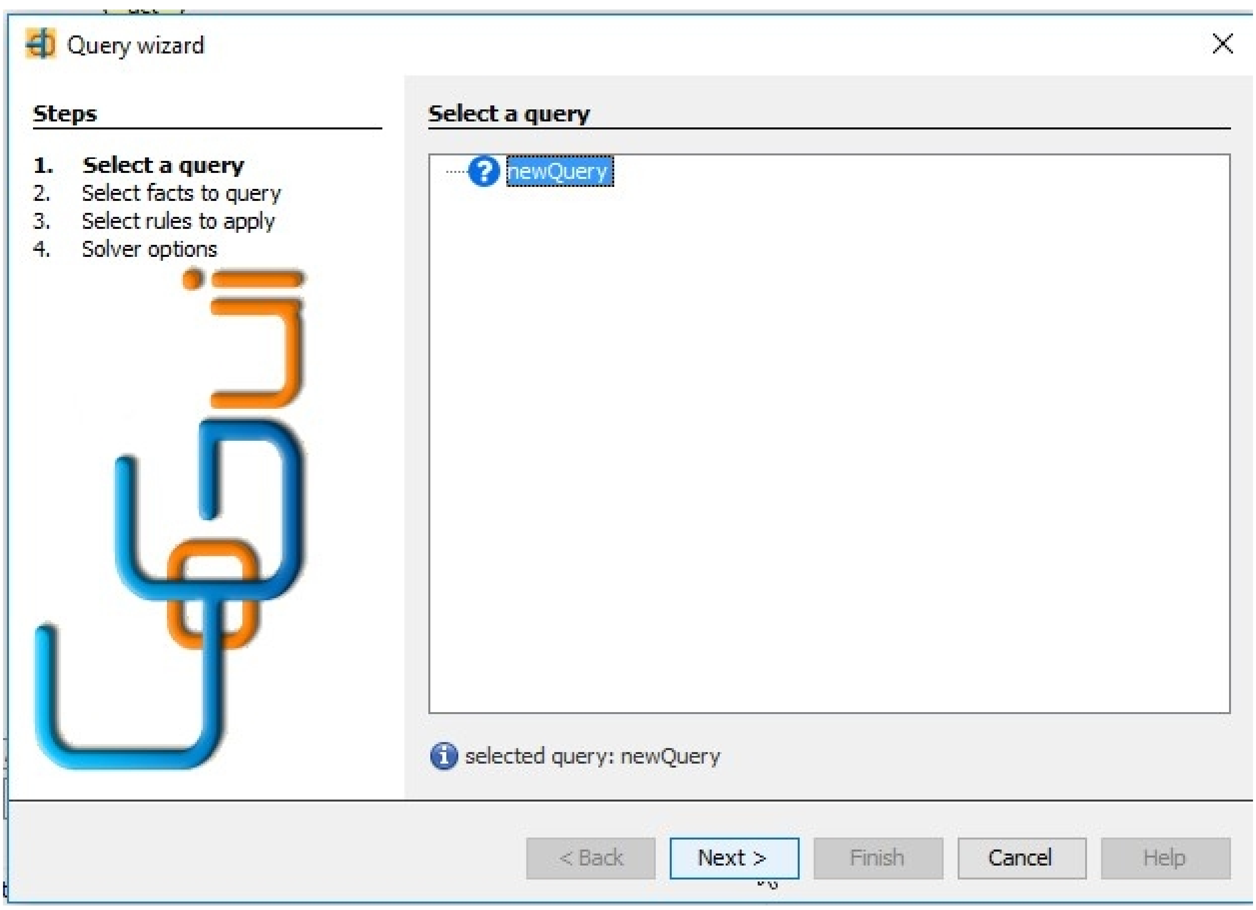
After the sum has been computed, the graph will be normalized and split when there is several connected components

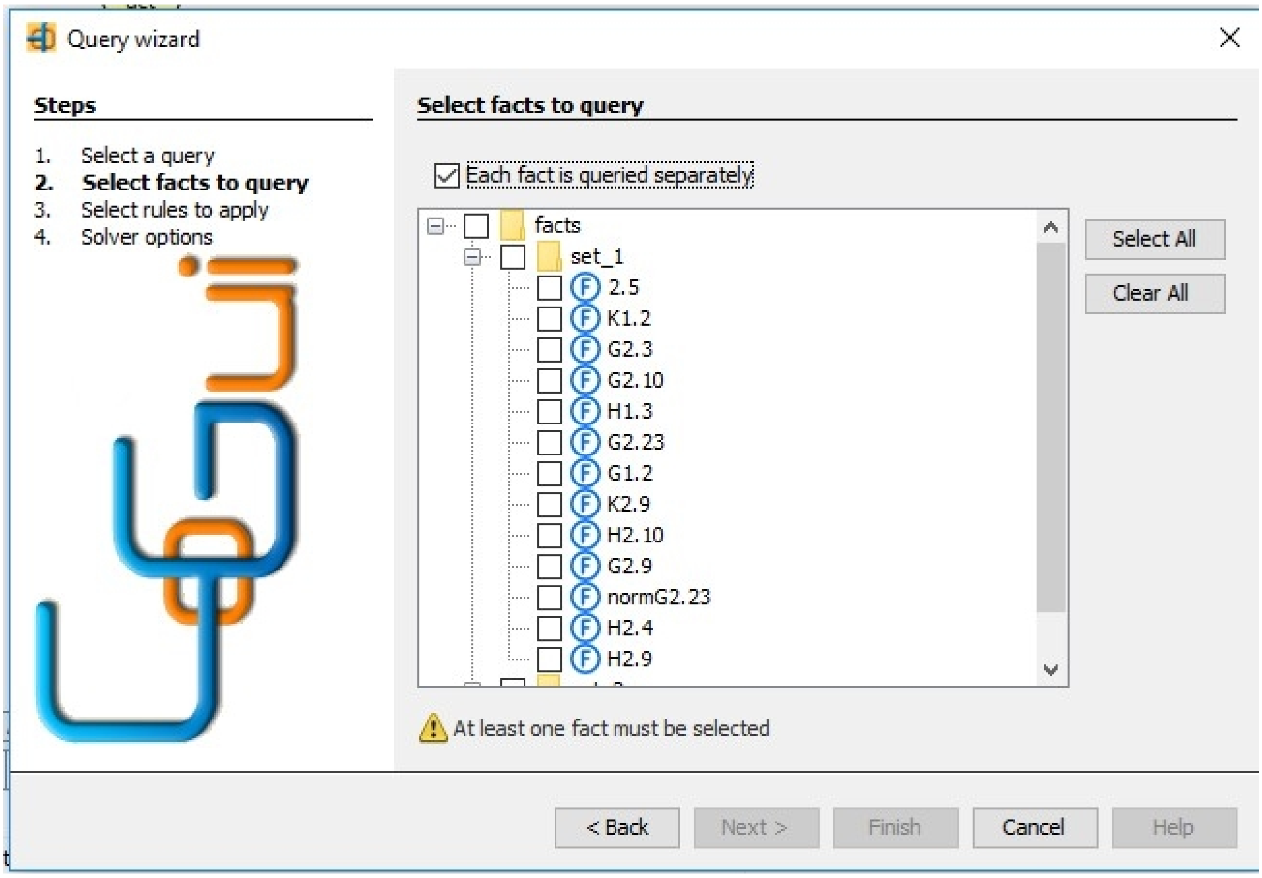
Querying

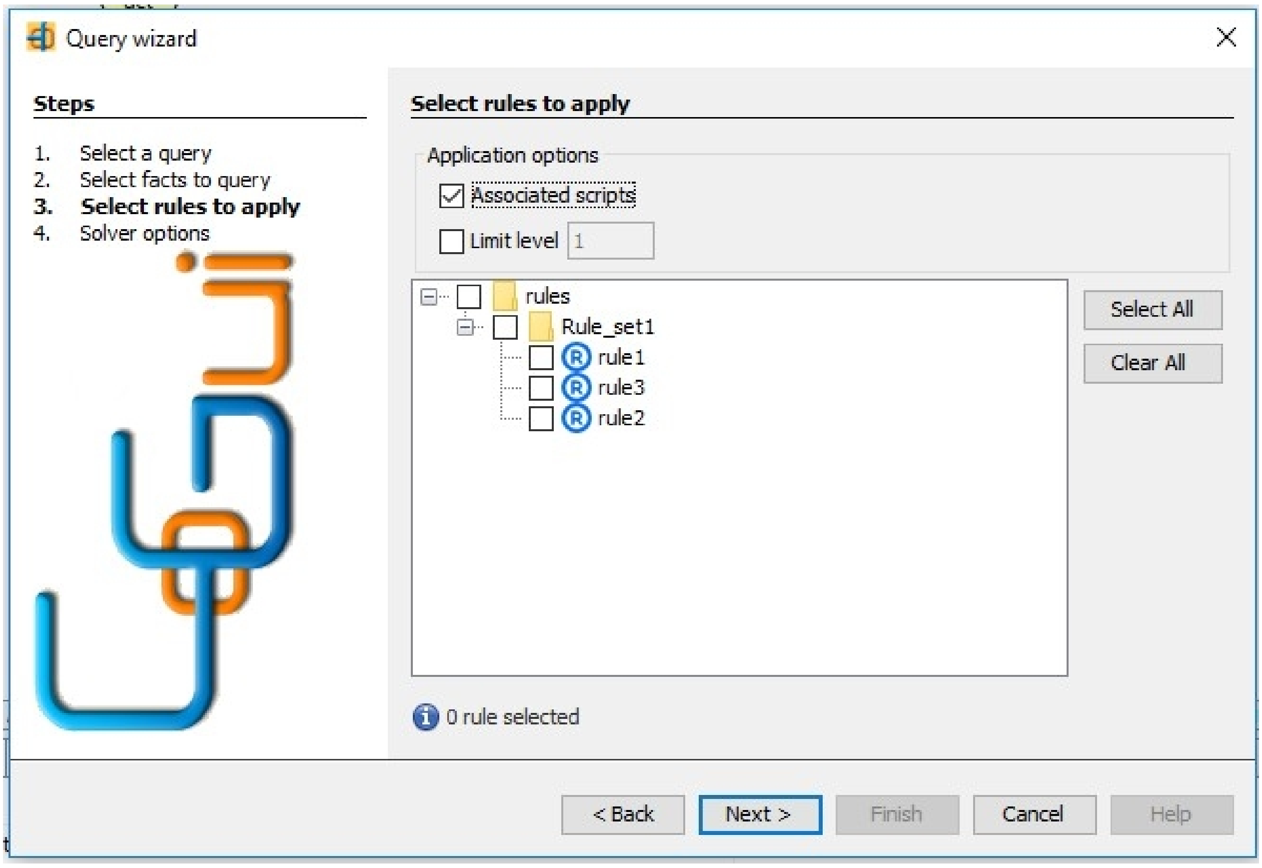
Launch the querying wizard

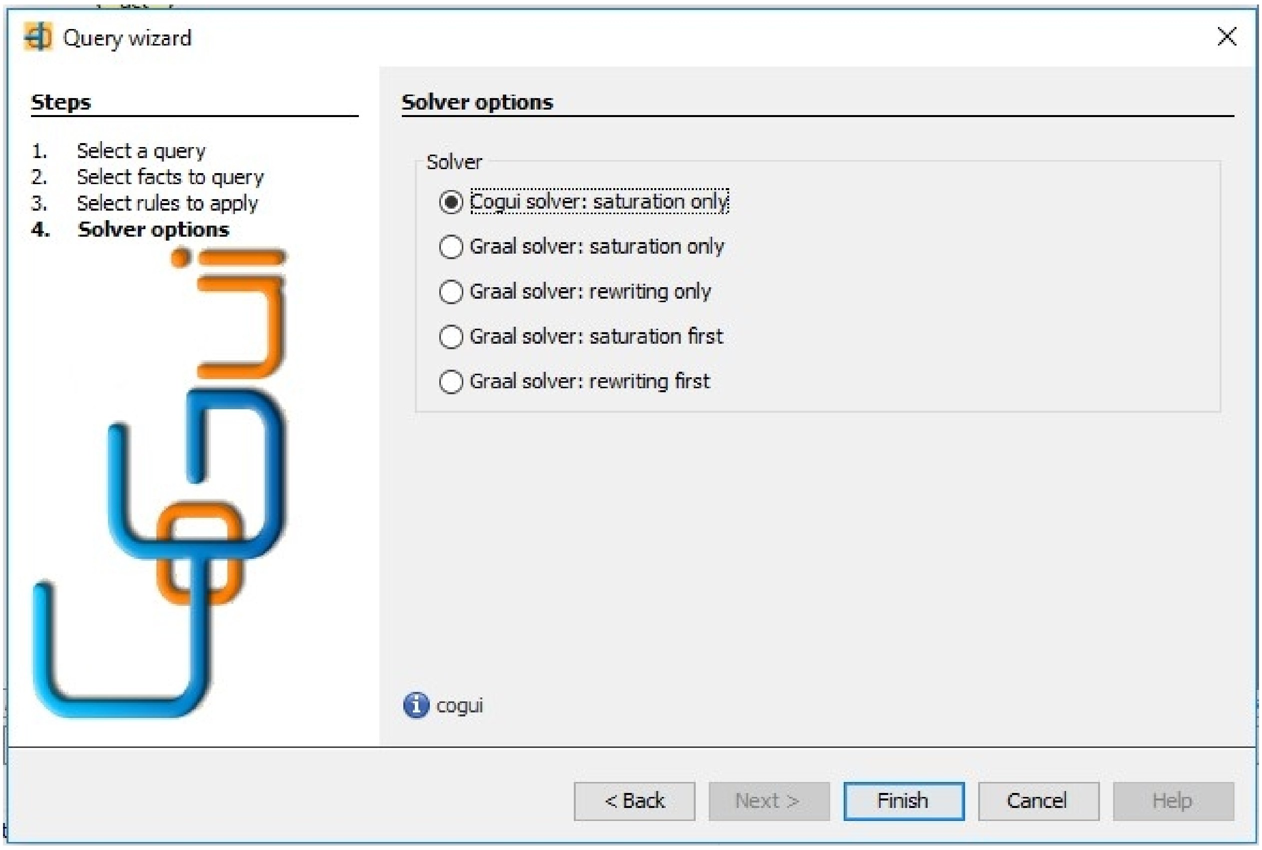


The four steps of the wizard:

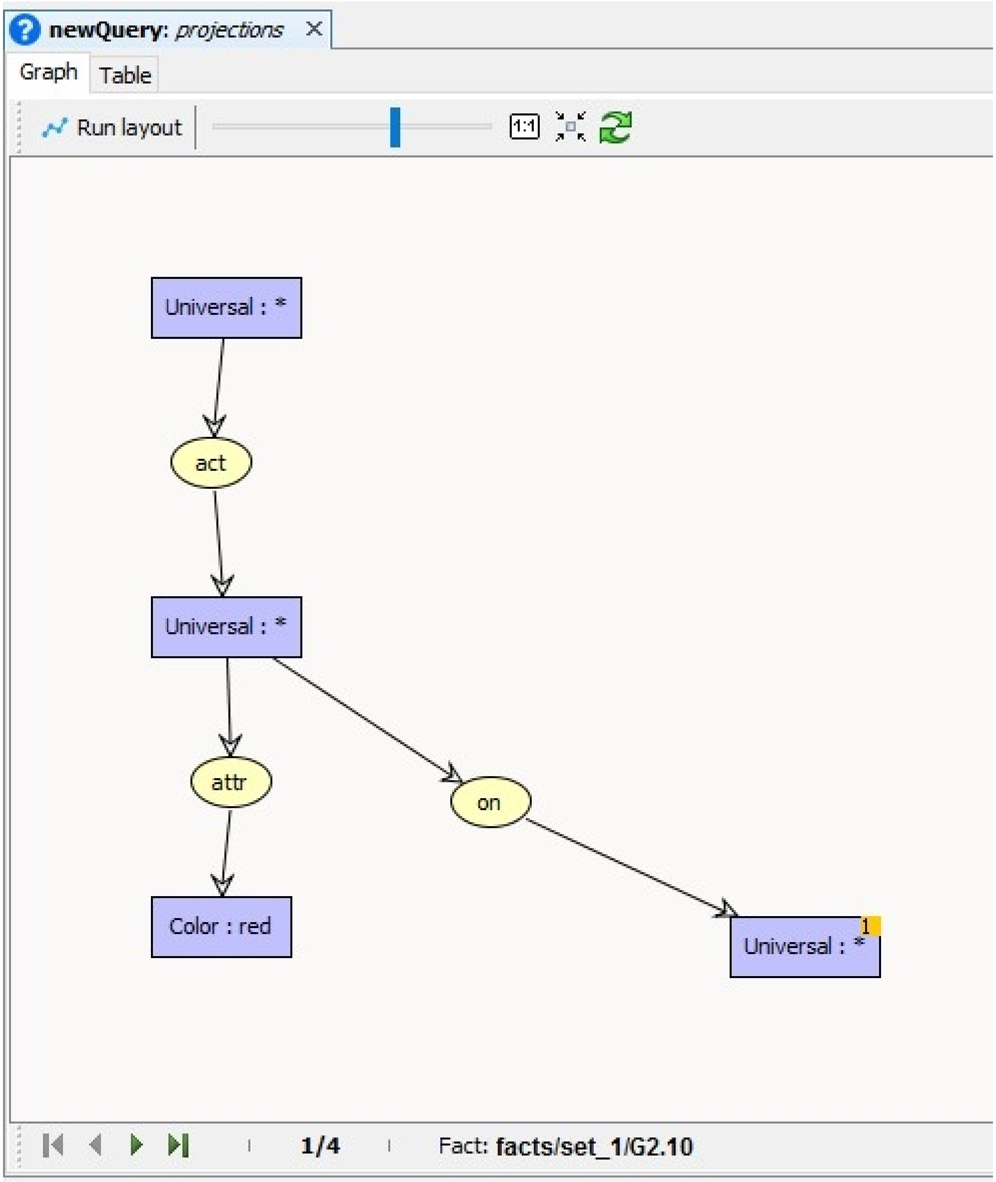



Query wizard
Steps
1. Select a query
2. Select facts to query
3. Select rules to apply
4. Solver options
Select facts to query
Each fact is queried separately
facts
set_1
2.5
K1.2
G2.3
G2.10
H1.3
G2.23
G1.2
K2.9
H2.10
G2.9
normG2.23
H2.4
H2.9
Select All
Clear All
At least one fact must be selected
< Back
Next >
Finish
Cancel
Help


Query wizard
Steps
1. Select a query
2. Select facts to query
3. Select rules to apply
4. Solver options
Select rules to apply
Application options
Associated scripts
Limit level
1
rules
Rule_set1
rule1
rule3
rule2
Select All
Clear All
0 rule selected
< Back
Next >
Finish
Cancel
Help



If CoGui solver is used, the results are displayed as graph



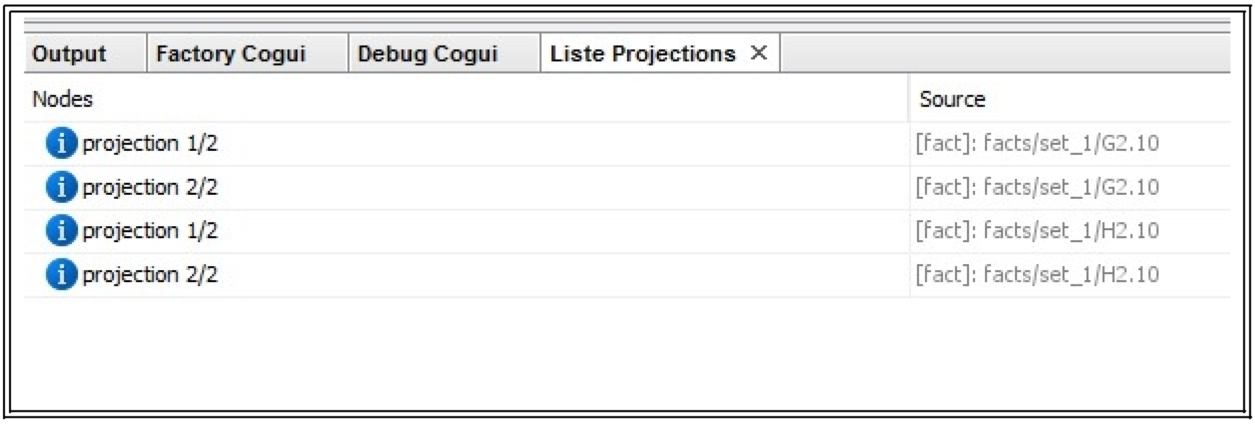
Results can also presented as list

newQuery: projections

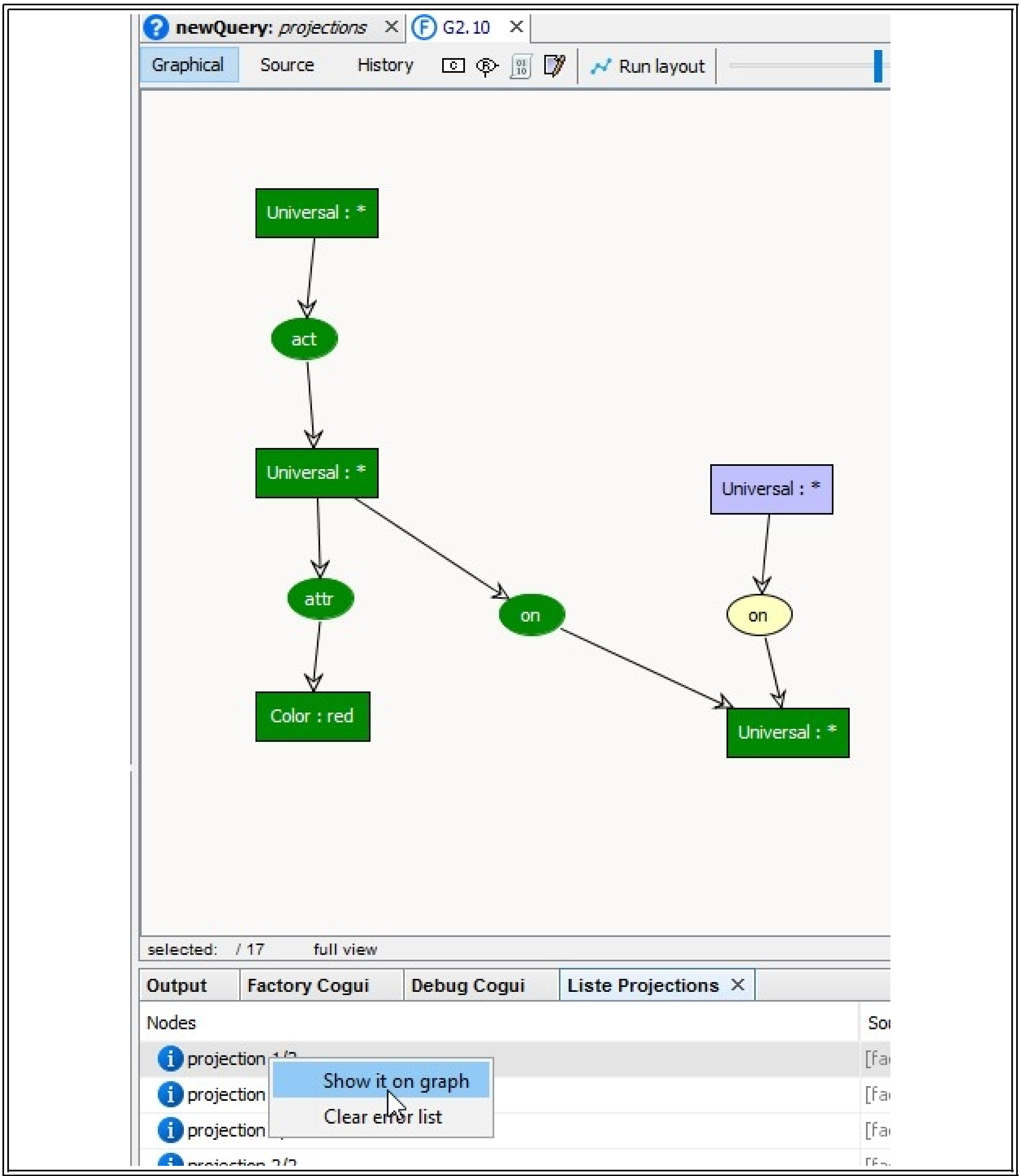
Graph | Table

Graph name	#Color_1	#Universal_2	#Universal_3	#Universal_4	#Universal_5
facts/set_1/G2.10	Color [red]	Universal [*]	Universal [*]	Universal [*]	Universal [*]
facts/set_1/G2.10	Color [red]	Universal [*]	Universal [*]	Universal [*]	Universal [*]
facts/set_1/H2.10	Color [red]	Universal [*]	Universal [*]	Universal [*]	Universal [Paul]
facts/set_1/H2.10	Color [red]	Universal [*]	Universal [*]	Universal [*]	Universal [*]

The list of projections are also provided in 'List projections view'



With messages from projection list it is possible to browse the facts



Import, Export and Convert

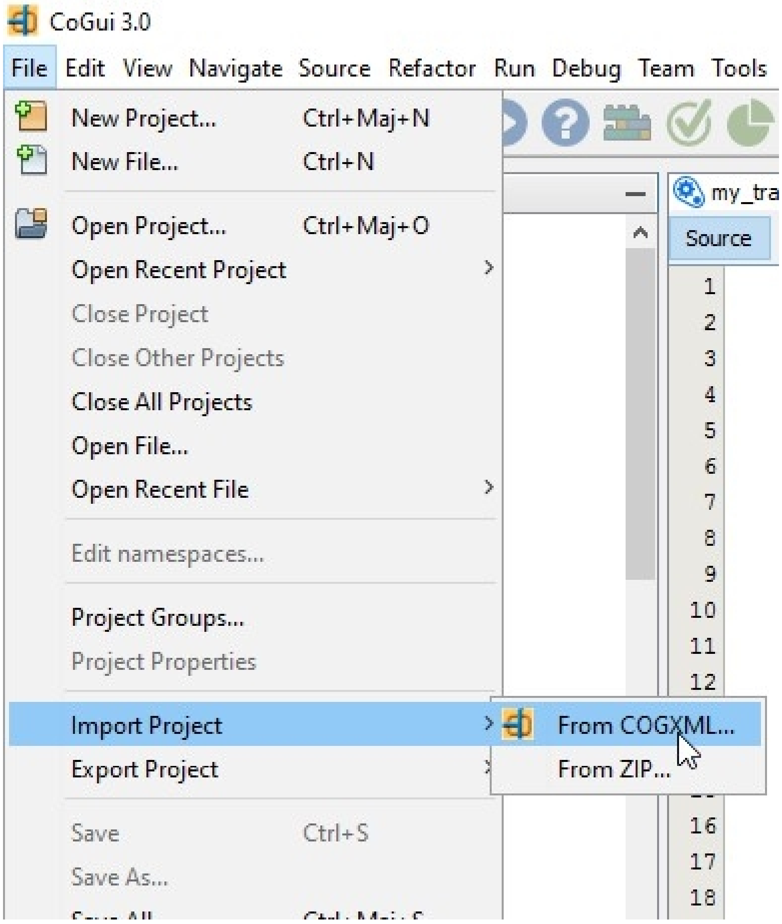
Users of previous versions of CoGui have the option of converting old projects to the new format. The COGXML format is not forgotten as it is also possible to export the projects again in this format. Read this section for more information To/From COGXML projects

CoGui propose several tools to exchange between different knowledge representation models:

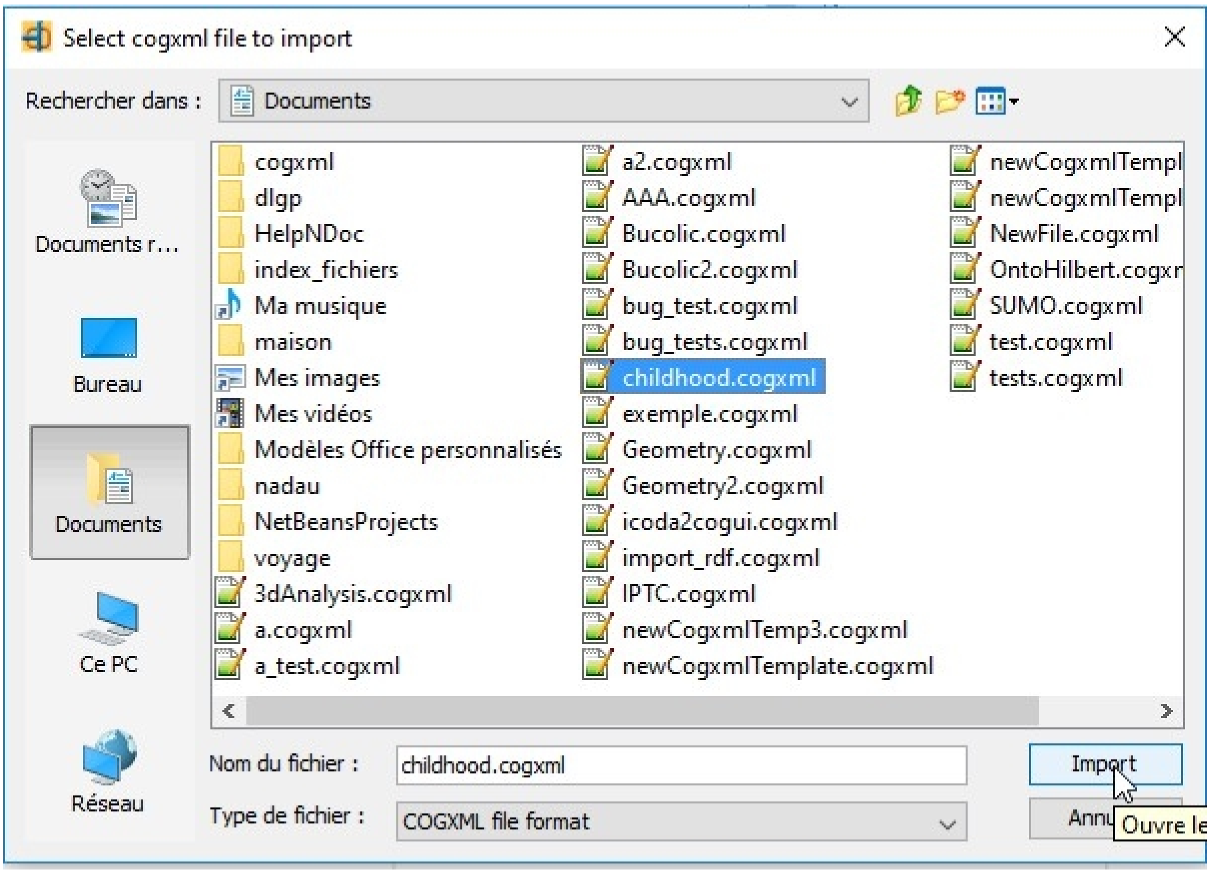
- To/From RDF(S) and OWL
- To/From Datalog±

To/From COGXML projects

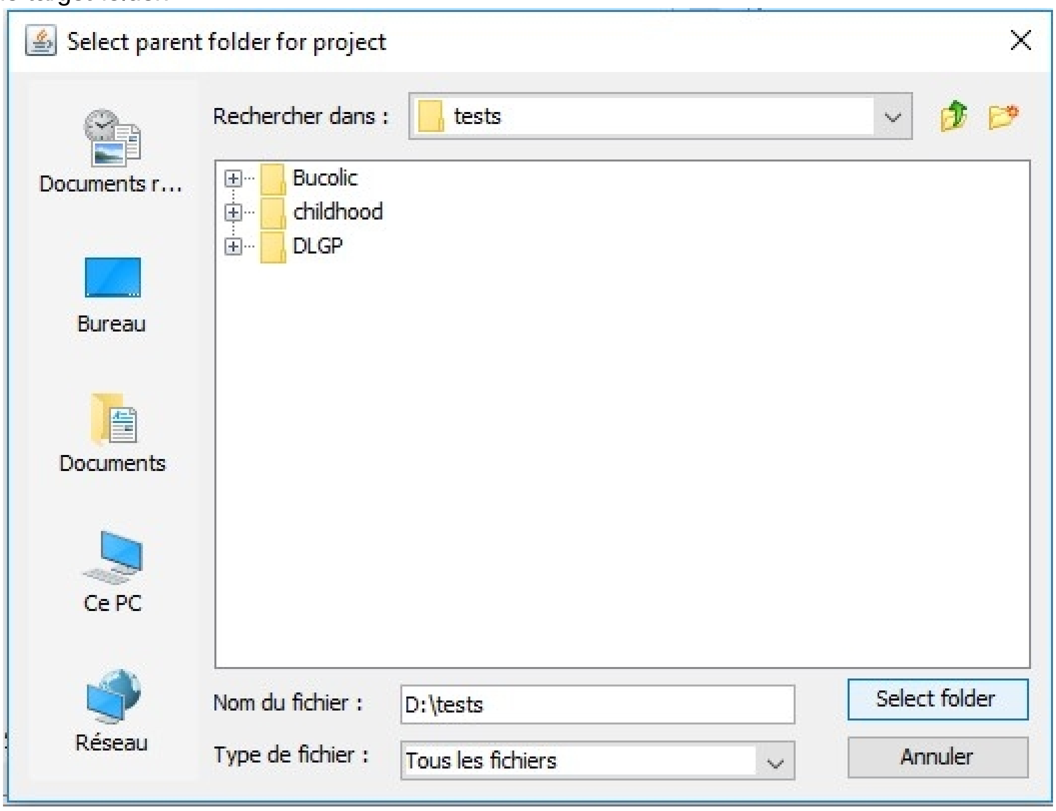
The menu option "File/Import projects" , in addition to the classic procedure, offers an action to import COGXML project.
COGXML projects can come from Cogitant project. It is also the form. This is also the format of previous versions of CoGui.



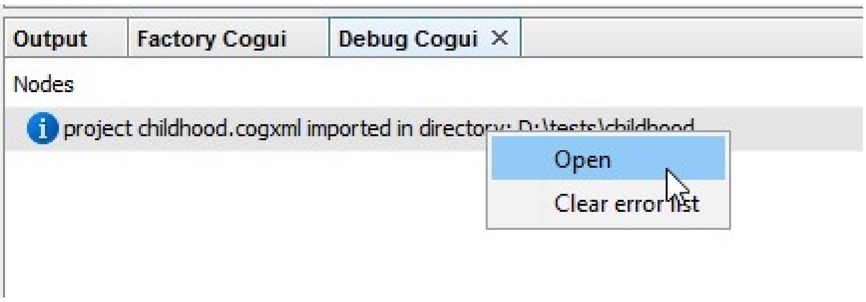
select to COGXML file to import:



select the target folder:



Then the new CoGui project can be opened from the message in Debug window:



To/From RDF(S) and OWL

CoGui is able to import RDF(S) documents.

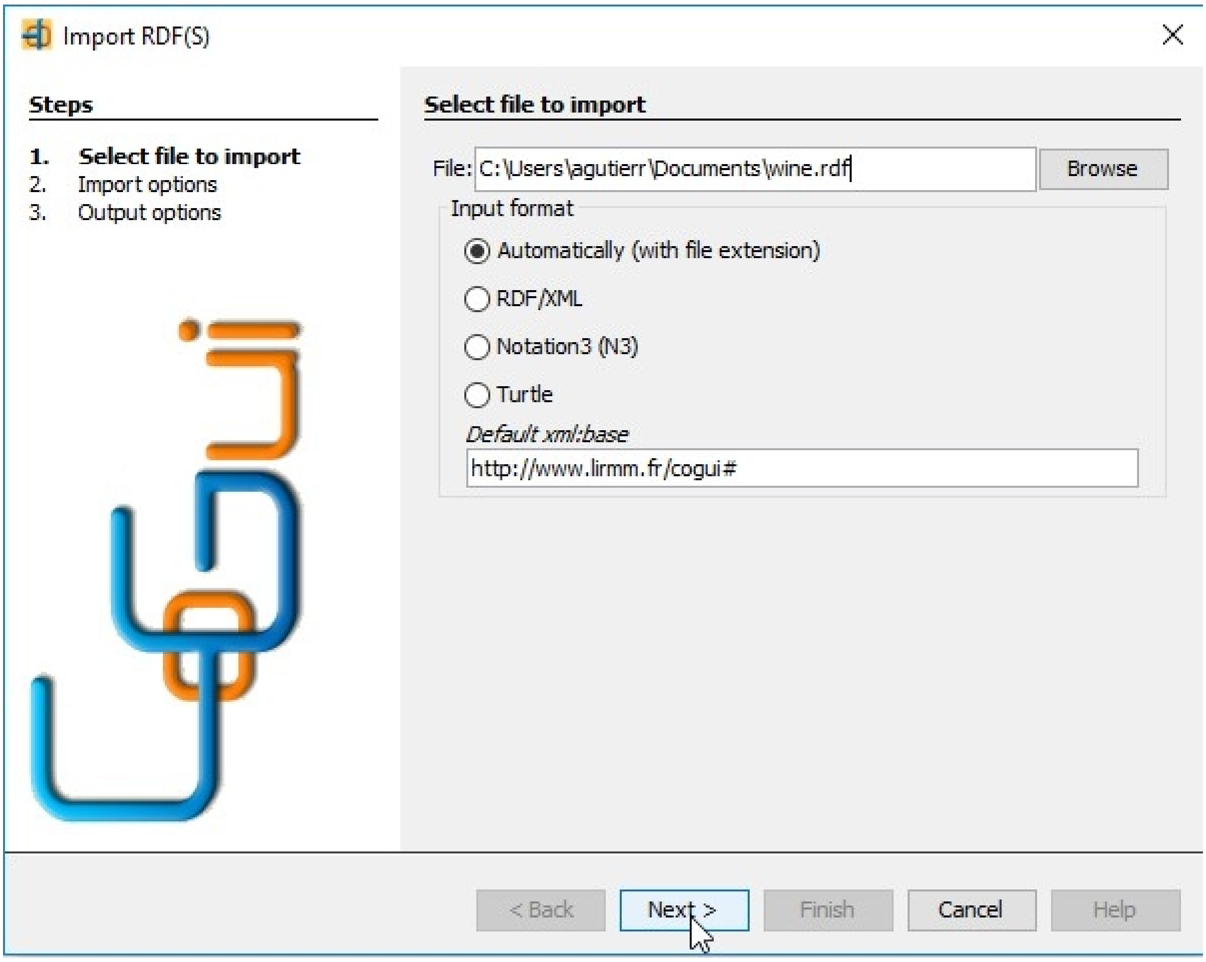
- The "natural" translation has the advantage of being natural and fully exploiting the CG features, but, on the other hand it does not apply to the whole RDF(S).
- The "raw" translation is sound and complete from a reasoning view point but is not visual nor a representation in the spirit of Conceptual Graphs (CGs).
- The new Graal translation use the Graal convertor from OWL to Datalog+ and then translates produced Datalog+ to CoGui

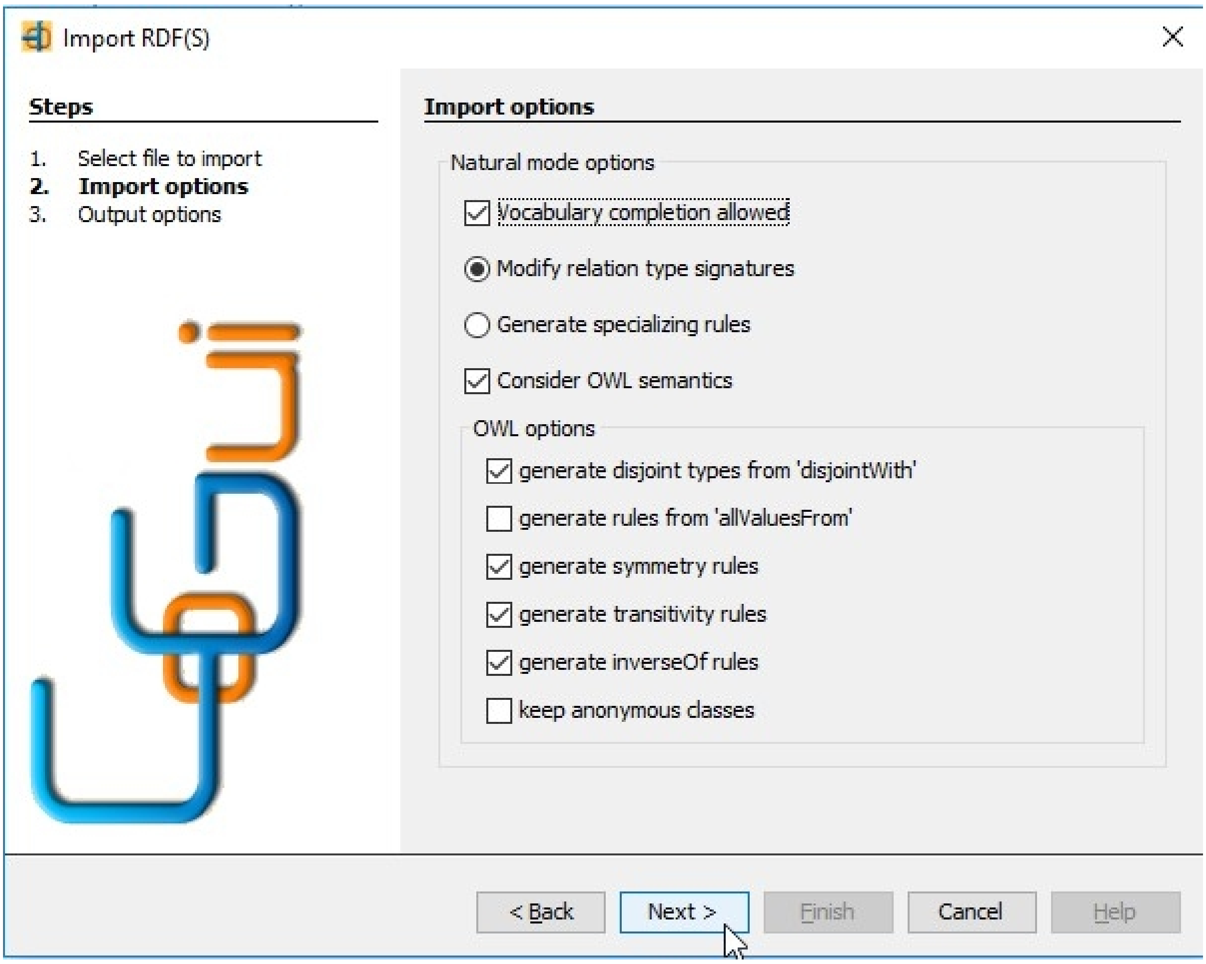
A tool to export in "natural" mode is also provided.

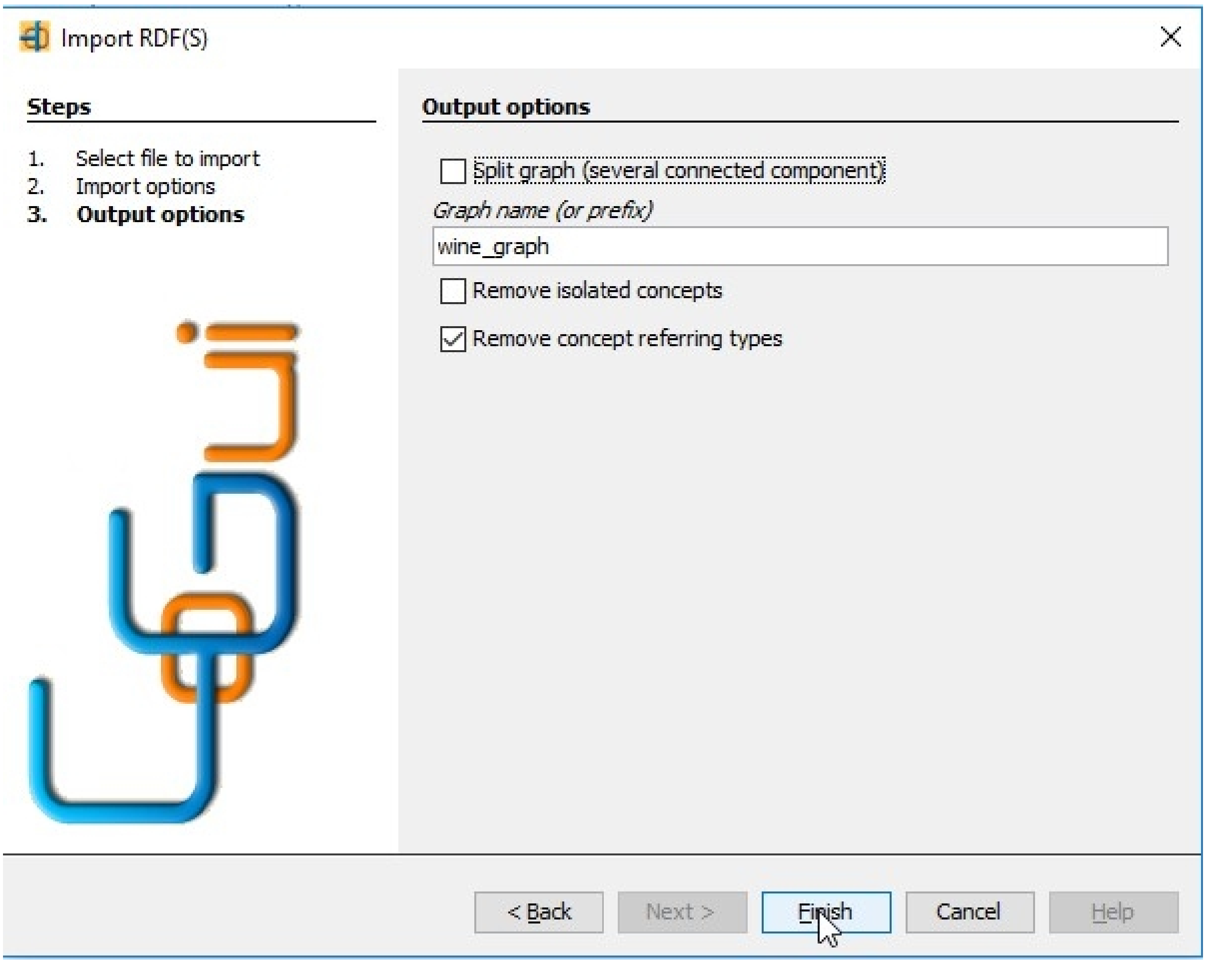
Import RDFS/OWL "natural" mode

This translation is intuitive and doesn't need rules to be applied because most of them are implicitly expressed in the support. The main idea behind it is to try to exploit as much as possible the separation between background knowledge and factual knowledge. Several options are available.

The 3 steps of the wizard:

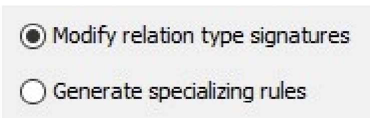

Import RDF(S)
Steps
1. Select file to import
2. Import options
3. Output options
Select file to import
File: C:\Users\agutierr\Documents\wine.rdf
Browse
Input format
Automatically (with file extension)
RDF/XML
Notation3 (N3)
Turtle
Default xml:base
http://www.lirmm.fr/cogui#
< Back
Next >
Finish
Cancel
Help


Import RDF(S)
Steps
1. Select file to import
2. Import options
3. Output options
Import options
Natural mode options
Vocabulary completion allowed
Modify relation type signatures
Generate specializing rules
Consider OWL semantics
OWL options
generate disjoint types from 'disjointWith'
generate rules from 'allValuesFrom'
generate symmetry rules
generate transitivity rules
generate inverseOf rules
keep anonymous classes
< Back
Next >
Finish
Cancel
Help



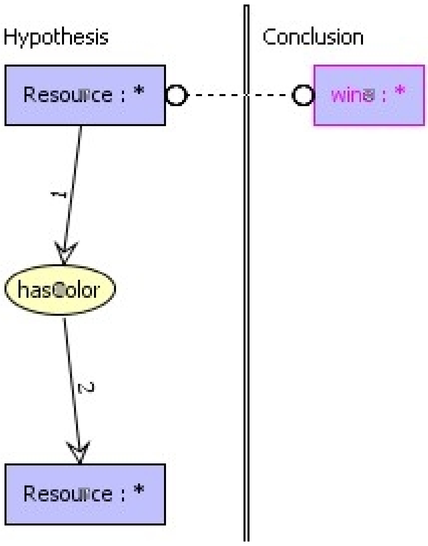
If support completion option is checked, support is enriched by the imports 

Uncheck the box to preserve current vocabulary and only take factual knowledge into account.

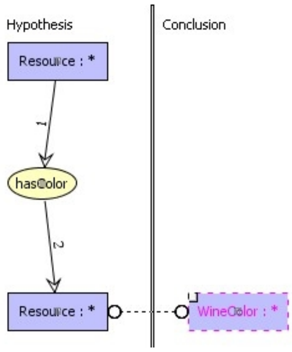
	When support completion is allowed two ways exist to express `rdfs:range` and `rdfs:domain` semantic. The default is to define relation type signatures but it can be expressed by range rules and domain rules, depending on the purpose. Do the user want to correct future factual imports or to have a way to verify imports with signature.

For example a property can be translated as a CG relation type with its signature or with rules:

signature



domain rule

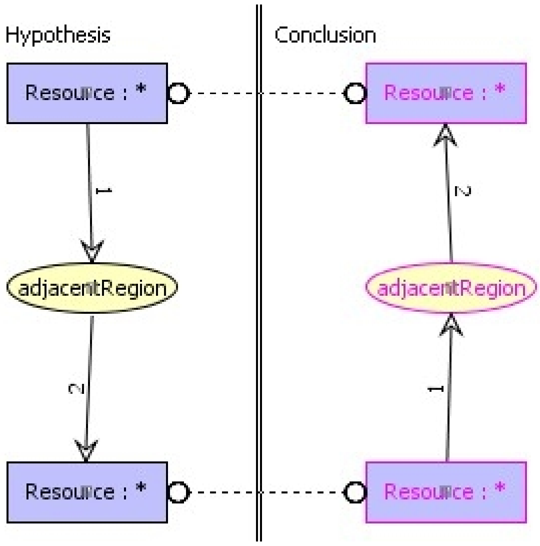


range rule

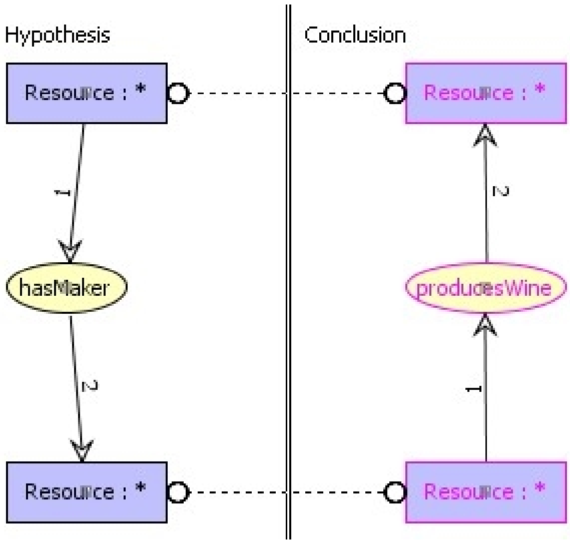
OWL Options

The semantics of RDFS is taken into account almost entirely. This is not the case with OWL. However, many ontologies are defined with OWL. In practice the processing of anonymous classes and the treatment of most used properties of OWL allows to recover a significant part of the semantics of these ontologies.

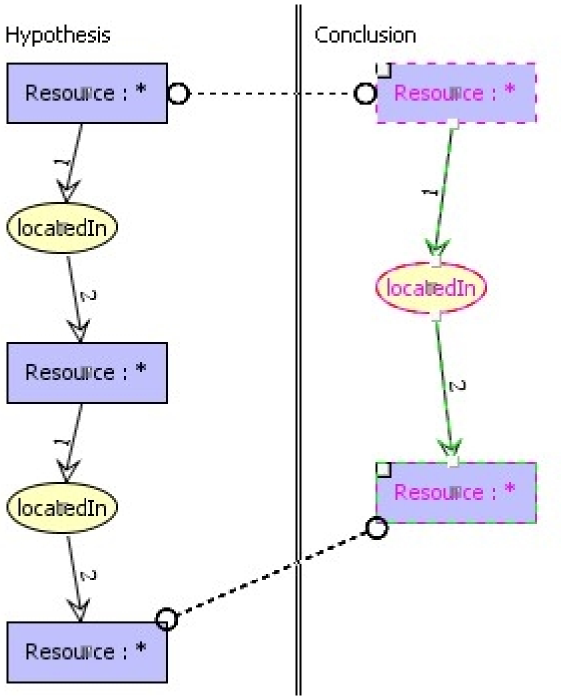
Fortunately ,a new import tool is provided by Graal extensions. See section Import RDFS/OWL with Graal
CoGui also offers an option that takes into account including anonymous classes, that are used during the processing of import. Some deductions were made using property restrictions, intersections and unions of classes and are expressed on the support. By default, after treatment all anonymous classes are removed from the type hierarchies, check option box to keep these classes.
The owl:disjointWith property is used to defined banned type in the concept type hierarchy.
In addition, some OWL properties are also expressed by rules, see below examples from importation of wine.owl ontology:



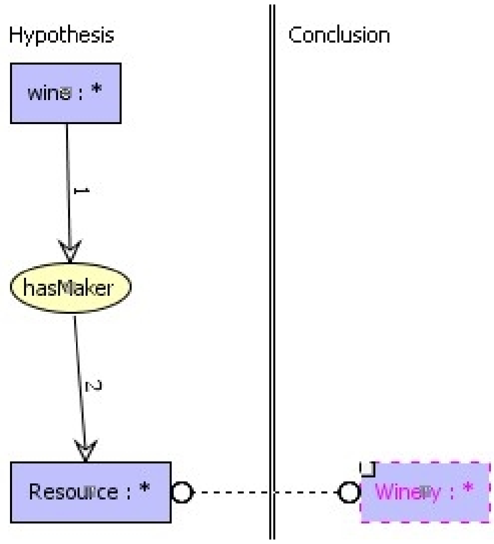
owl:SymmetricProperty



owl:inverseOf



owl:TransitiveProperty

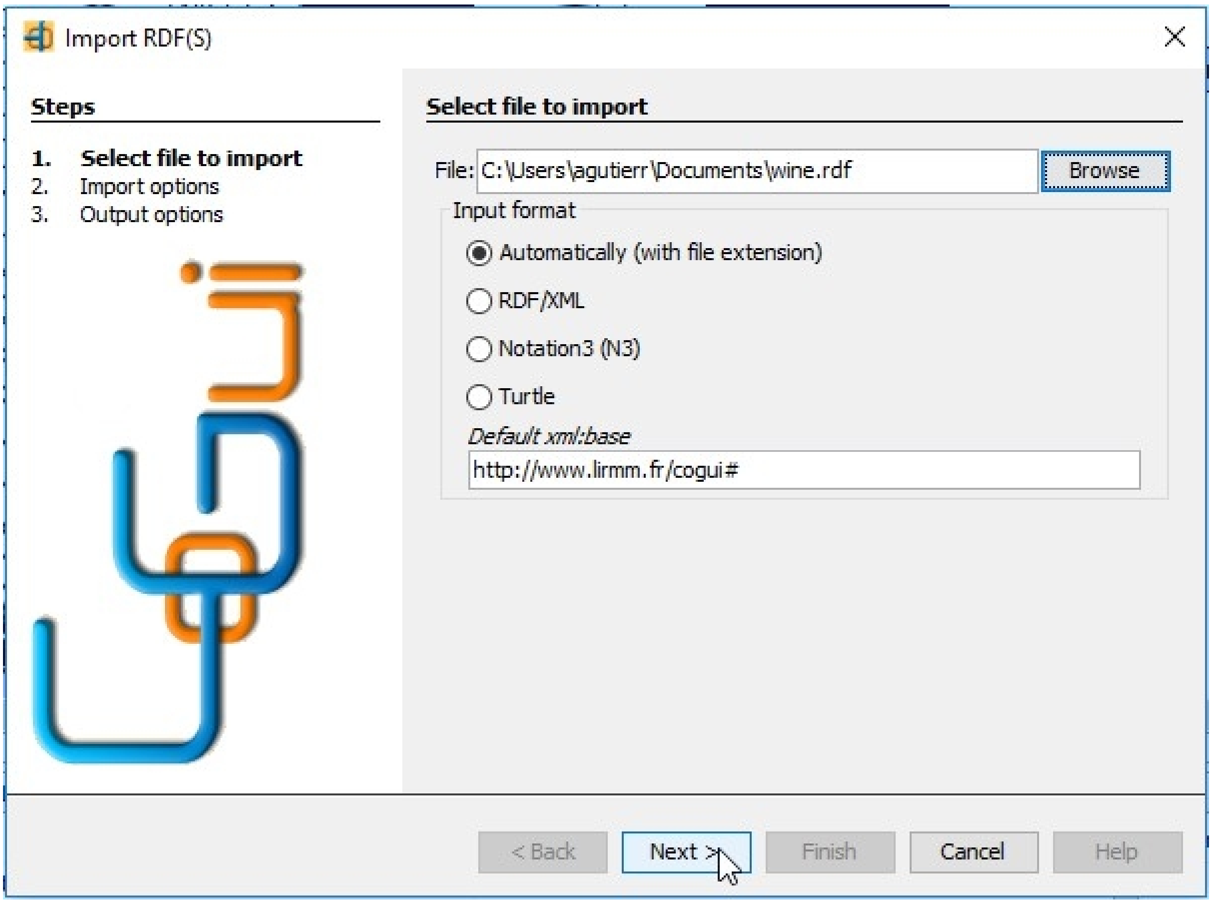


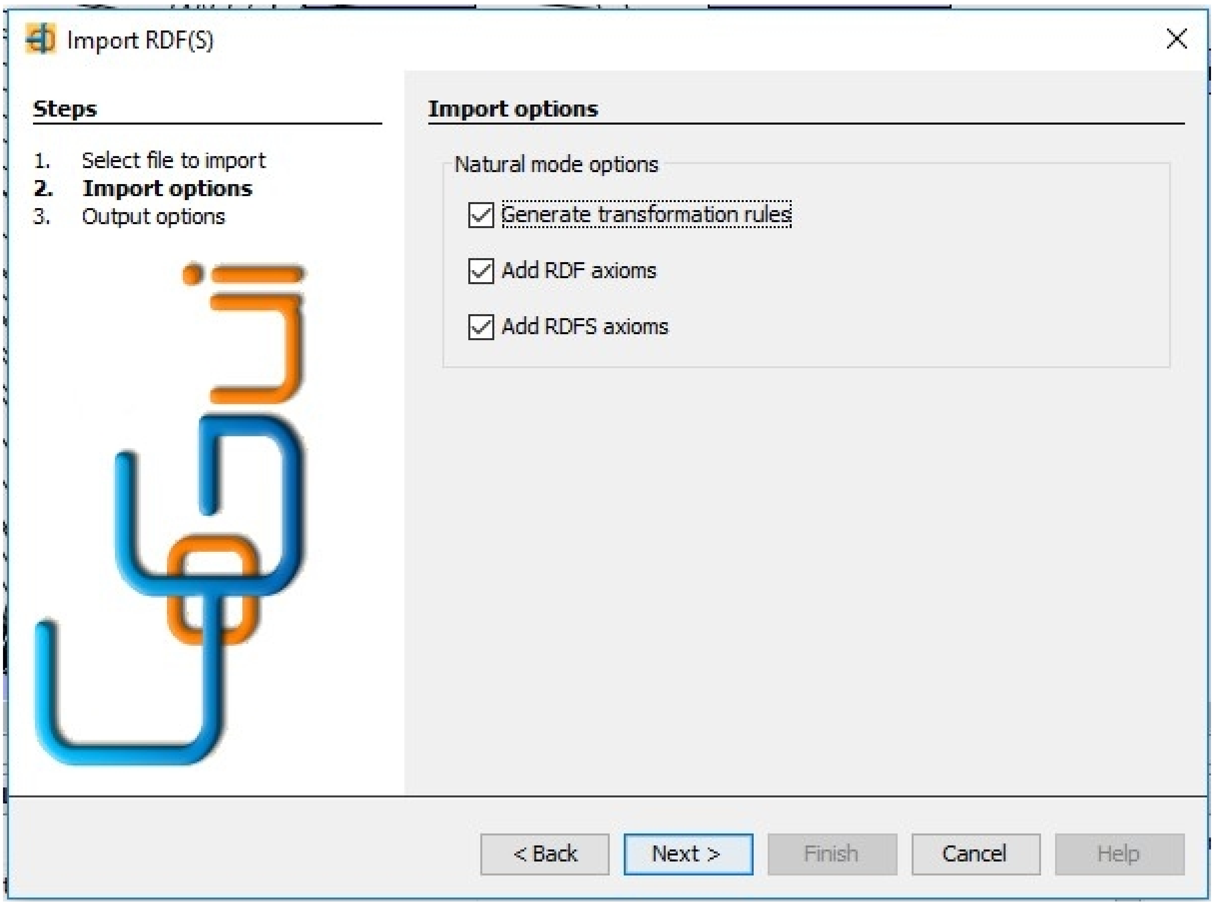
owl:allValuesFrom

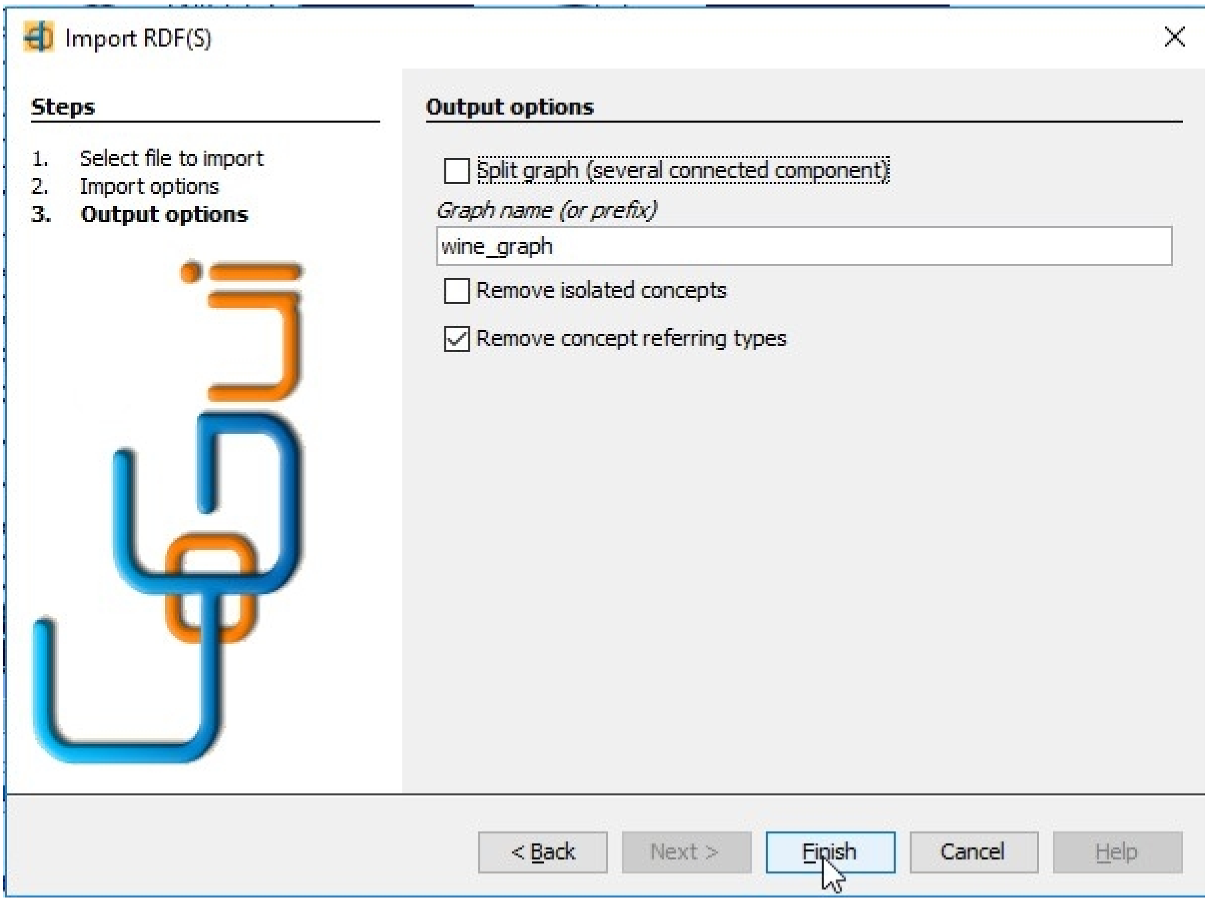
Import RDFS "raw" mode

The "raw" translation simply translate each triple RDF in a ternary relation where each of the concept nodes of the relation will represent the RDF triple elements. It ensures soundness and completeness of homomorphism. However, this translation is not visual or in the spirit of Conceptual Graphs as such (the support is flat).

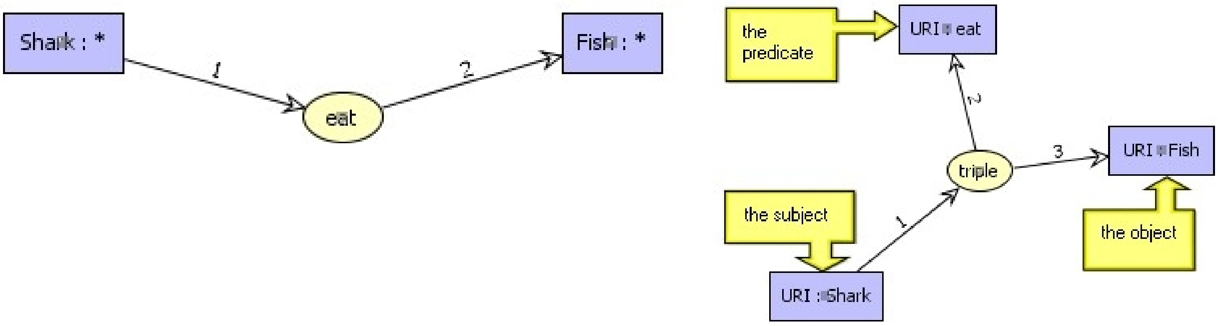
3 steps to import:


Import RDF(S)
Steps
1. Select file to import
2. Import options
3. Output options
Select file to import
File: C:\Users\agutierr\Documents\wine.rdf
Browse
Input format
Automatically (with file extension)
RDF/XML
Notation3 (N3)
Turtle
Default xml:base
http://www.lirmm.fr/cogui#
< Back
Next >
Finish
Cancel
Help

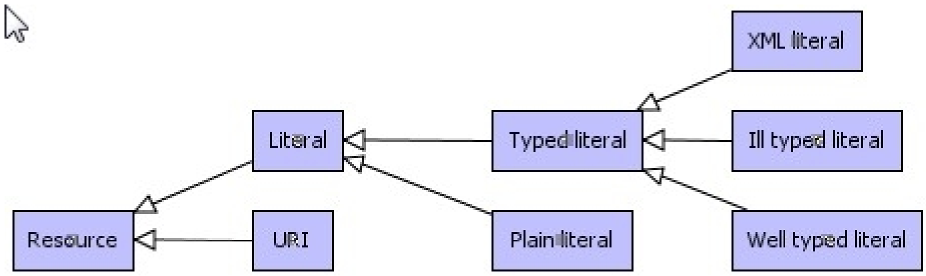

Import RDF(S)
Steps
1. Select file to import
2. Import options
3. Output options
Import options
Natural mode options
Generate transformation rules
Add RDF axioms
Add RDFS axioms
< Back
Next >
Finish
Cancel
Help

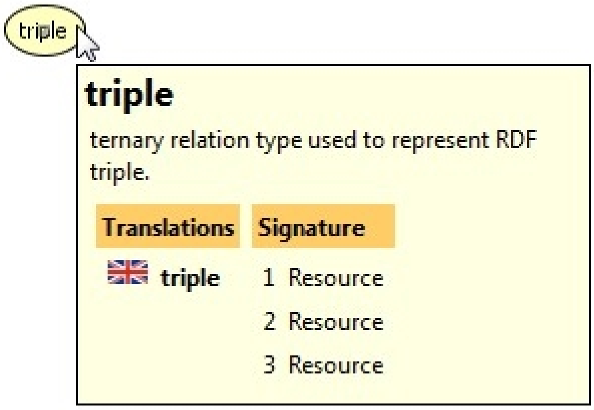


The raw translation of a triple:

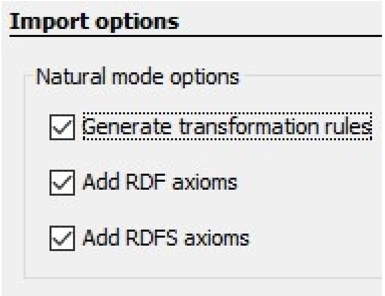


raw mode use a predefined vocabulary:

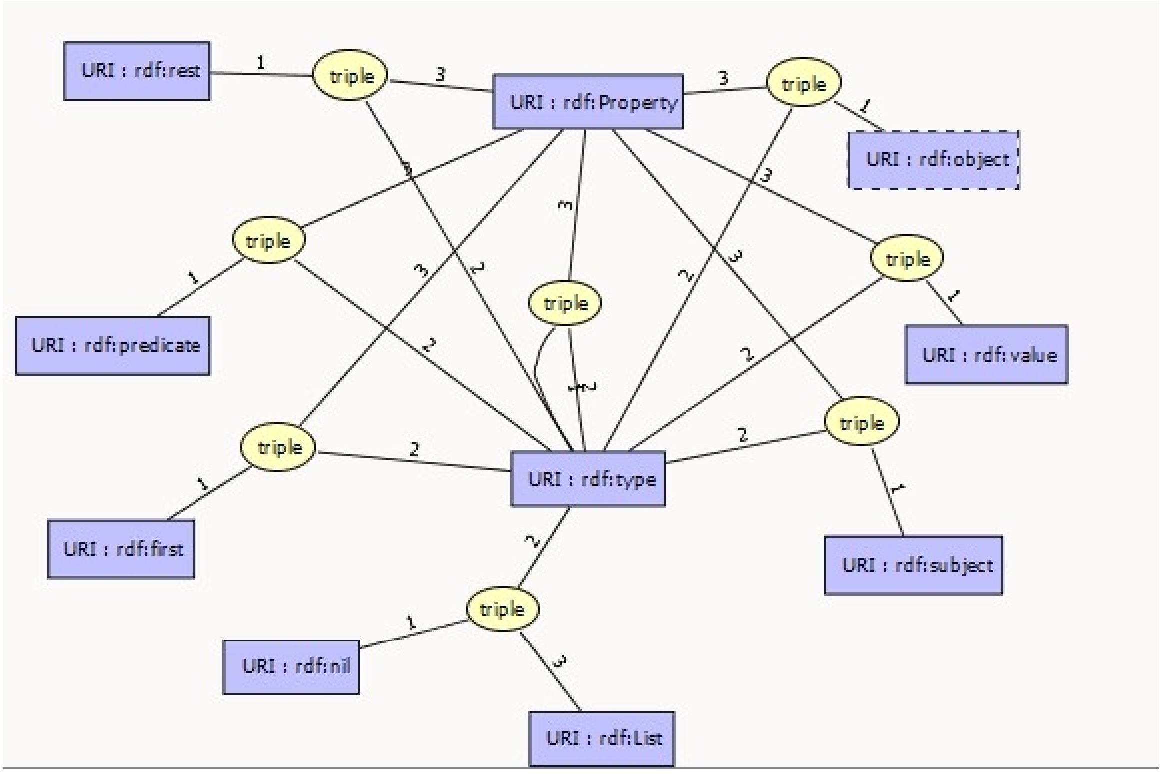




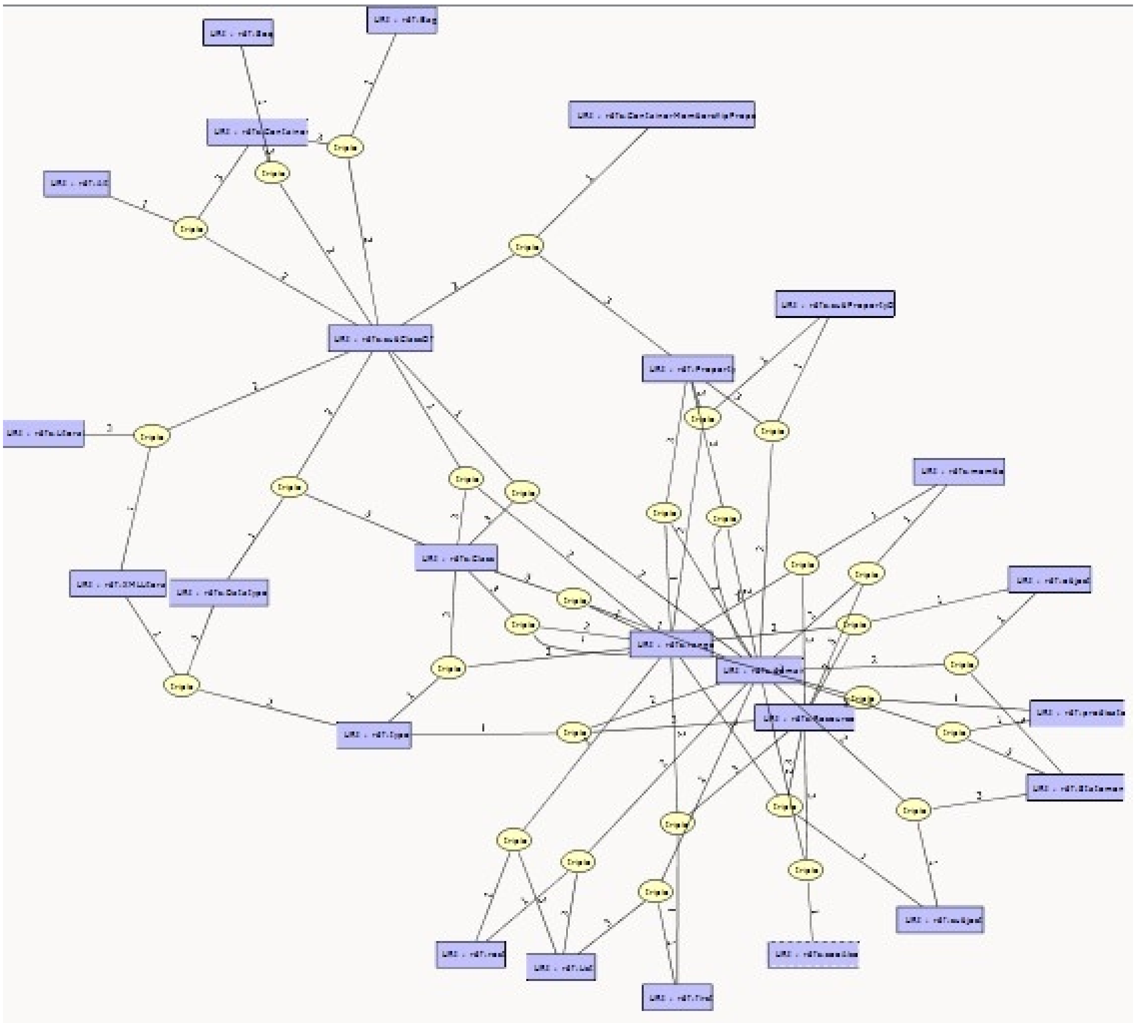
This mode is interesting to demonstrate some properties but in practice, it cannot be used with big graphs because the application of all rules on the sum graphs of imports plus rdf(s) axioms quickly increases the graph size. To be used to find homomorphisms, this mode requires to add RDF and RDFS axioms as facts and to apply several RDF and RDFS rules. This facts and rules can be automatically created during importation. Check corresponding boxes on the wizard option panel:



RDF axioms:


URI : rdf:rest
triple
URI : rdf:Property
URI : rdf:object
URI : rdf:predicate
URI : rdf: value
URI : rdf:first
URI : rdf:type
URI : rdf:subject
URI : rdf:nil
URI : rdf:List

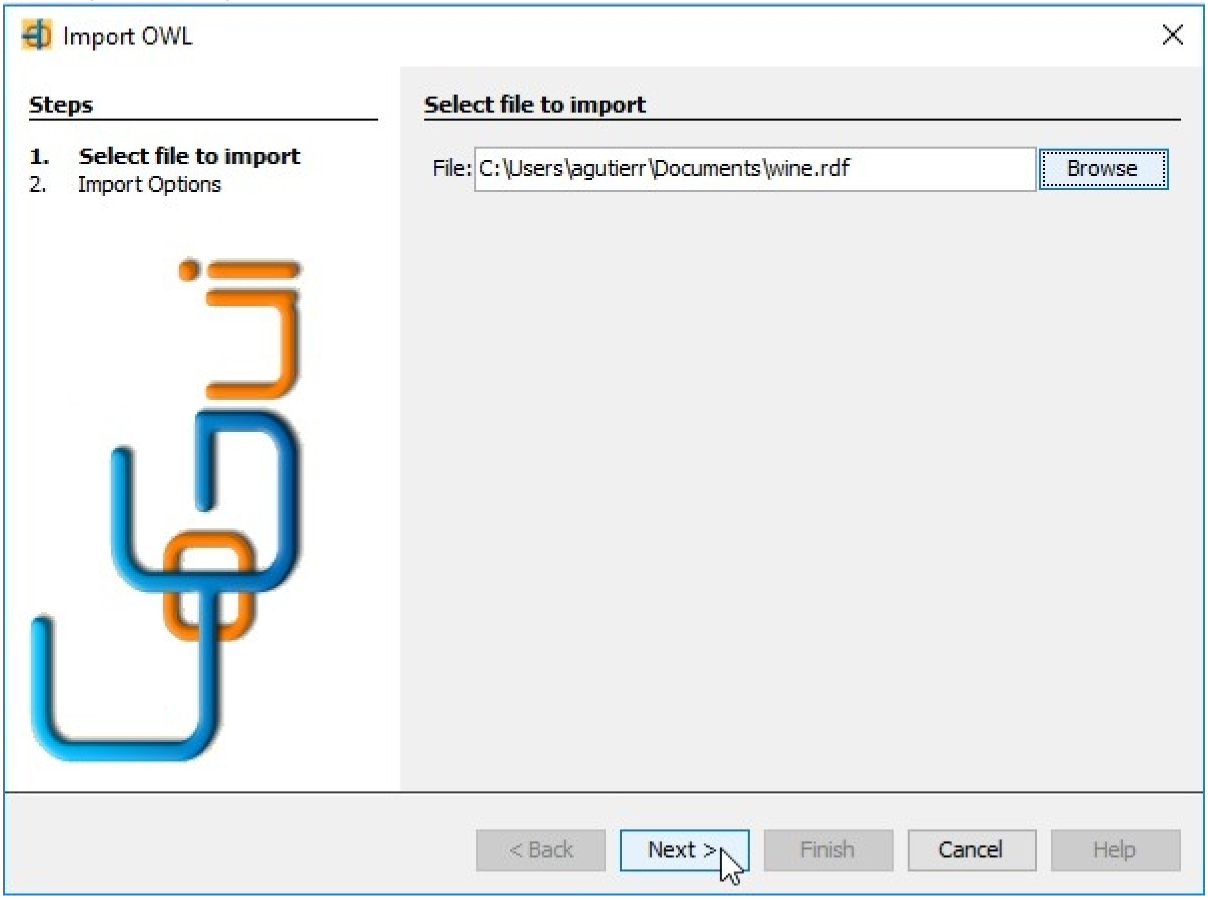
RDFS axioms:

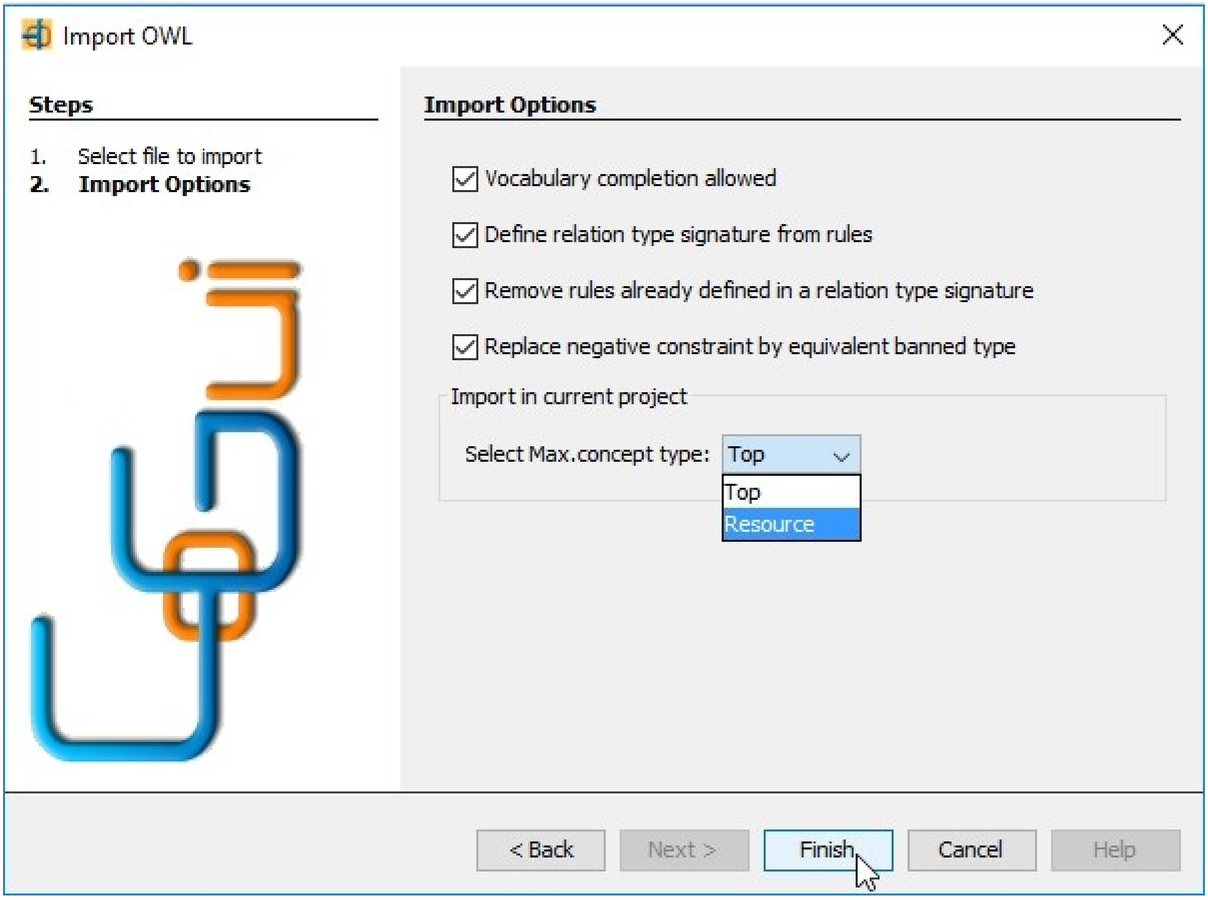


Import RDFS/OWL with Graal

[Graal Java toolkit](). provides a translator from any OWL2 ontologies to Datalog+: [OWL2DLGP](). CoGui wizard runs OWL2DLGP then use Datalog+ translation to import ontology to CoGui model.

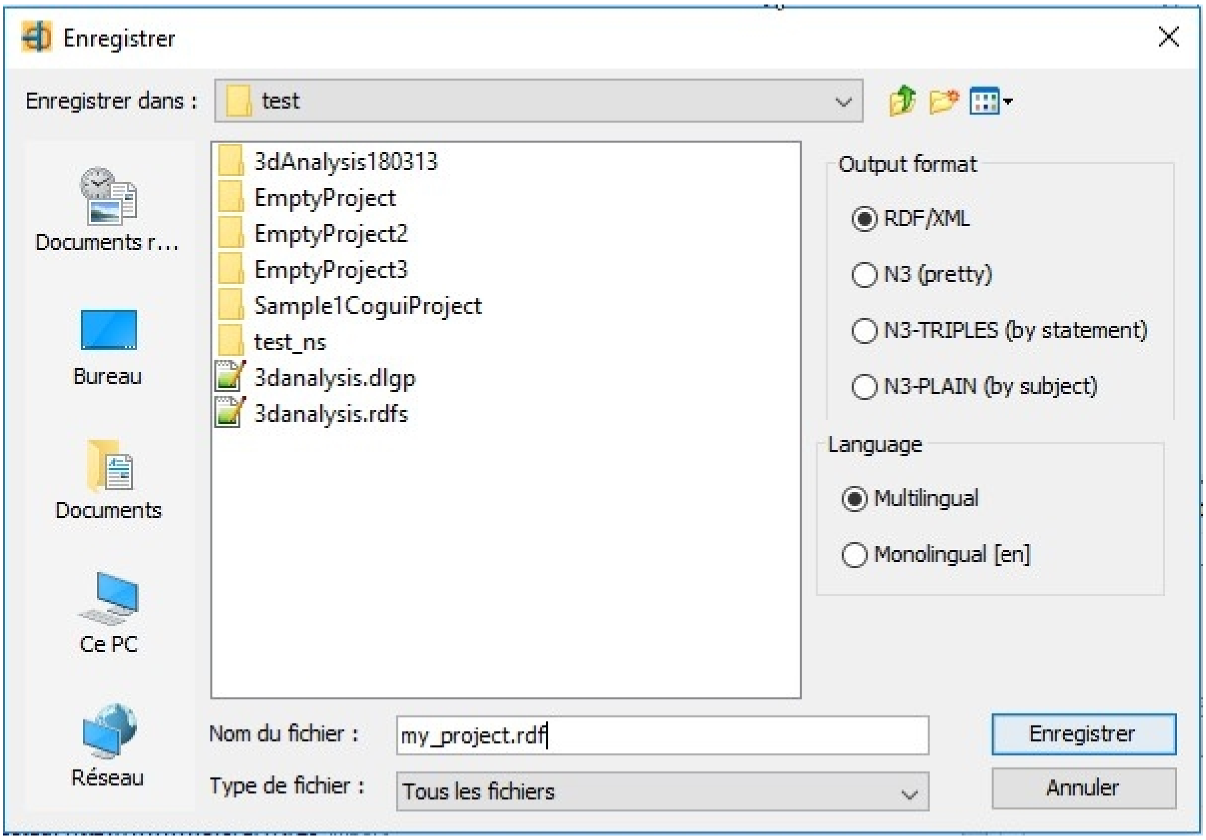
The 2 steps of the import wizard:





Export RDFS "natural" mode

Cogui is able to export projects to RDF with RDF/XML or N3 formats. For now only type hierarchies and facts can be exported. Rules , constraints and type disjunctions are ignored. But domain and range constraints induced by relation type signature are translated. All namespaces defined in the project are declared in output. By default, a multilingual version is produced, a monolingual version is also available with current selected language.



To/From Datalog±

DLGP (for Datalog Plus) is a textual exchange format at once human-friendly, concise and easy to parse.
CoGui provides an editor for this language with syntax highlighting and a navigator.
Switch to graphical and DLGP representation with the factory view is a good way to become familiar with the language. See section The factory view .
DLGP files can also be imported. See section Import from Datalog±
Finally, you can also export the whole project. See section Export to Datalog±

The format can be seen as an extension of the commonly used format for plain Datalog. Datalog± may define four kinds of knowledge elements:

- Facts
- Existential rules
- Negative constraints
- Conjunctives queries

As usually in Datalog, variables begin with an upper-case letter and constants with a lower-case letter. We distinguish between regular constants(called constants hereafter) and literals, which are values belonging to some datatype. Literals are given as double-quoted strings or numeric values(integers and floats).
The file name has the extension .dlgp or .dlp. Character encoding is assumed to be UTF-8.
Complete syntax is described in this paper: DLGP: An extended Datalog Syntax for Existential Rules and Datalog±Version 2.0

Some examples of different elements are available here.

Some examples

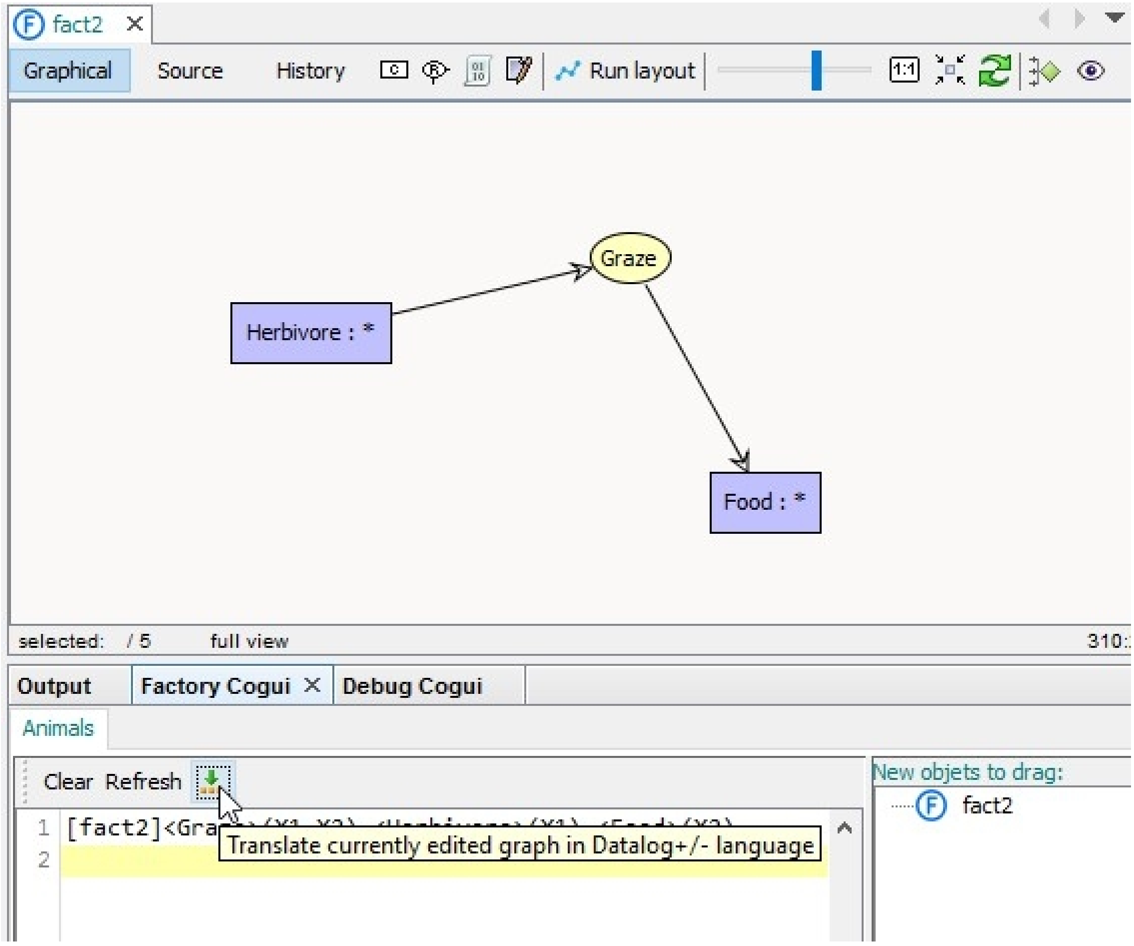
Datalog± expressions	Cogui conversion

`[a_fact]fatherOf(zeus,apollo), god(zeus), god(apollo).`	Every kinds of knowledge elements can be named with simple strings .
`fatherOf(<Zeus>,<Apollo>), god(<Zeus>), god(<Apollo>).`	Zeus can not be used as a constant name in DLPG language.
`belongsTo(<Zeus>,<Greek+pantheon>).`	Use + symbol to represent a space in the Datalog constant name.
`?:-fatherOf(X,apollo).`	Is there someone father of Apollo ?
`distance(athens,marathon,42.195), town(athens),town(marathon).`	A ternary predicate with a literal value(float).
`distance(athens,marathon,"42km195"), town(athens),town(marathon).`	A ternary predicate with a literal value(string).
`distance(athens,marathon,42) ,town(athens),town(marathon).`	A ternary predicate with a literal value(int).
`siblingOf(Y,X):-siblingOf(X,Y).`	A rule to define a symmetric relation.
`parentOf(X,Z):-parentOf(X,Y),parentOf(Y,Z).`	A rule to define a transitive relation.
`god(Y),human(Z), parentOf(Y,X),parentOf(Z,X):-demigod(X).`	A rule with new variables in conclusion (head of the rule). Unlike safe clauses in classic Datalog, Datalog± accepts new variables in the rule heads.
`X=Y:-equals(X,Y).`	A rule with equality as a conclusion. Head (conclusion) part of the rule can contains one or more equality relations between frontier concepts. They are represented as coreference links on the cogui model.

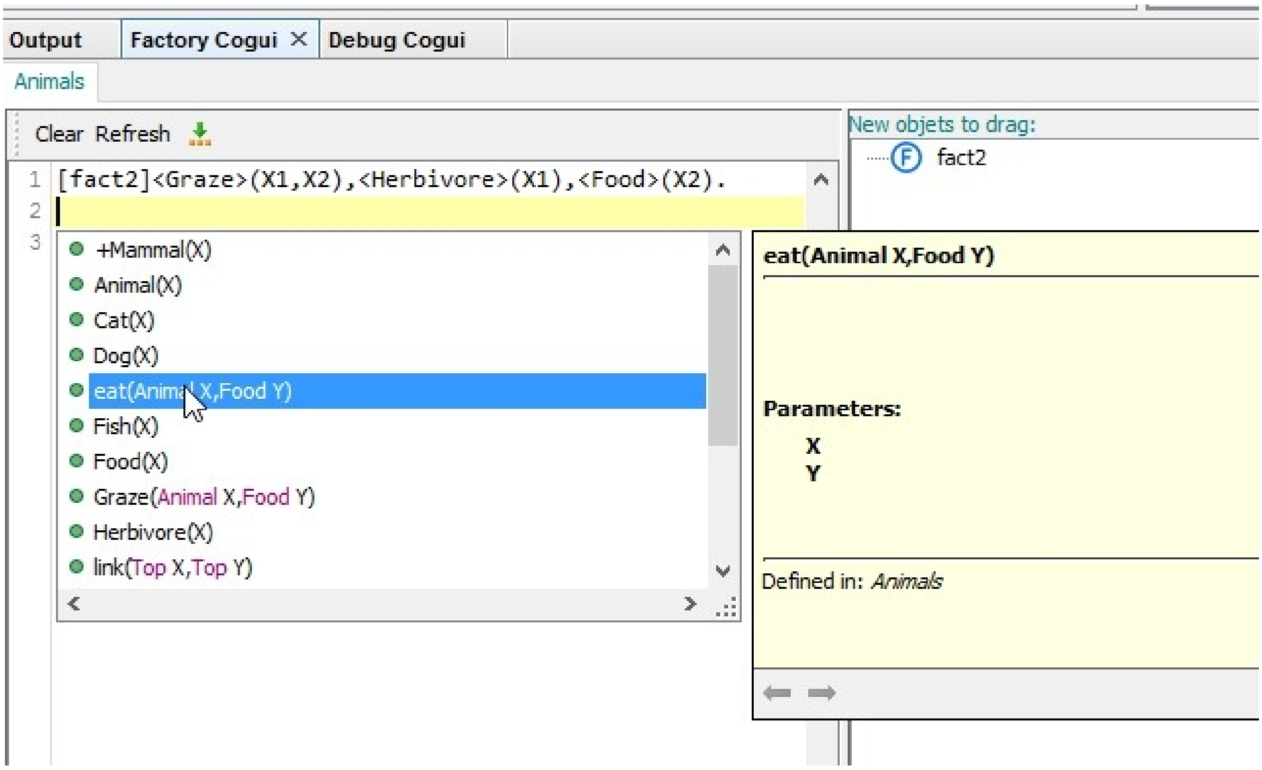
The factory view

Its brevity makes language datalog± an effective tool for the construction of new objects in the knowledge base, while the graphical representation is preferable to read (visualize) or update these objects. The "Factory" view is intended to allow concomitant use of both representations. It can immediately translate a text datalog± in its graphic equivalent and reciprocally translate any object of the knowledge base in its datalog± translation. Note that the factory view features concern facts, rules, queries and constraints but not the vocabulary. Classic import/export must be used for this purpose.

Datalog and graphical representations are complementary. With Factory view you can switch back and forth between the two.
Button on toolbar translates from currently edited object to datalog:

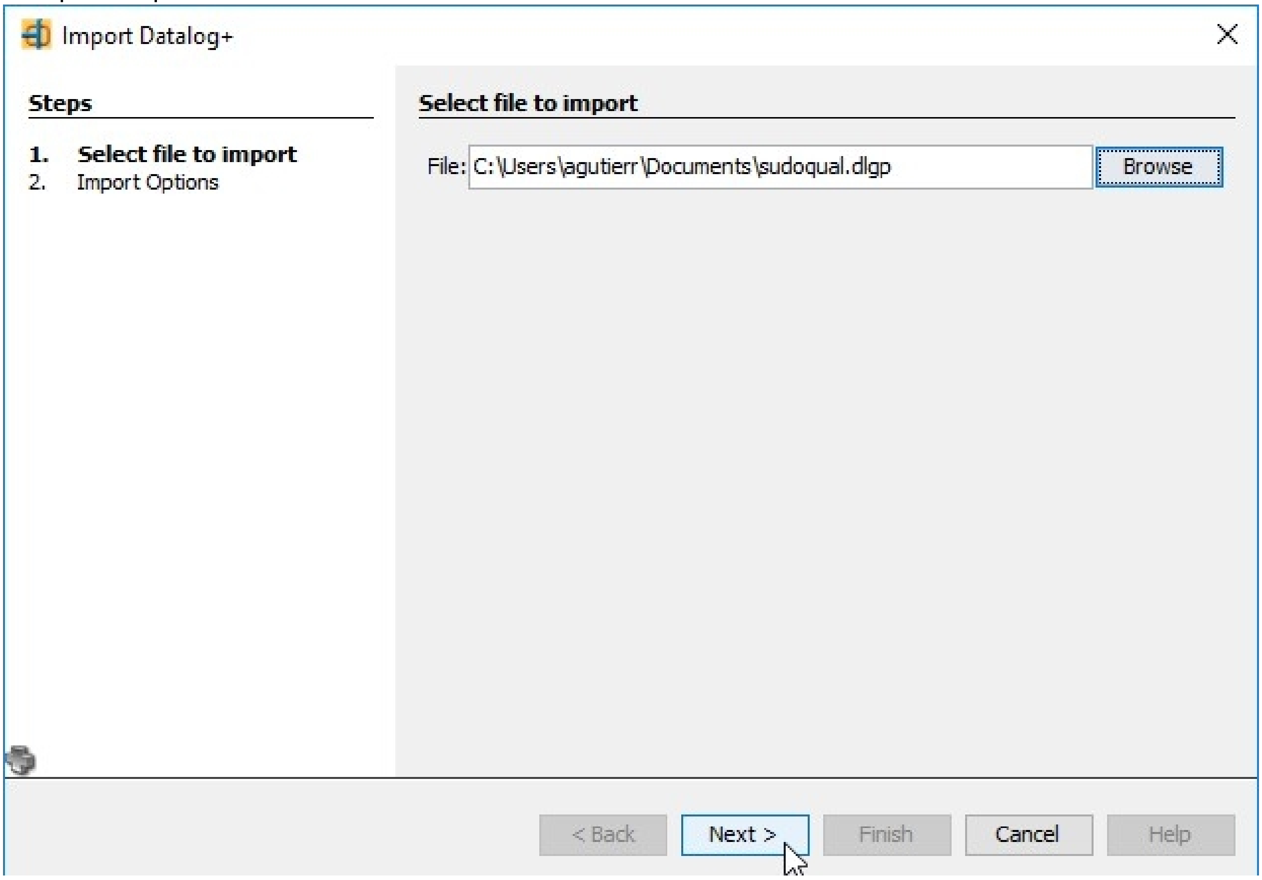


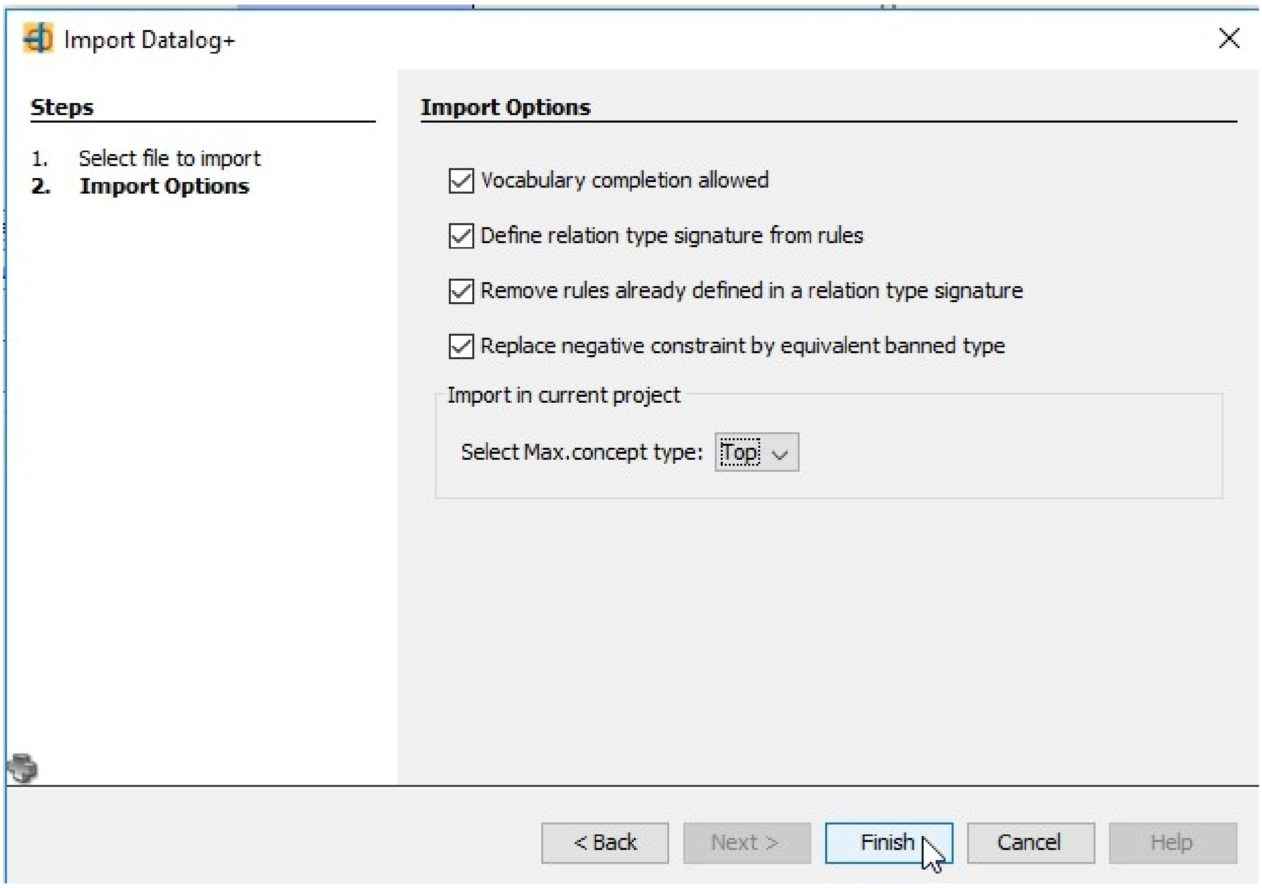
And right panel list all currently objects described in dlgp editor.
Press **Ctrl-Enter** to trigger the completion tool inside the DLGP editor.



Import from Datalog±

2 steps to import DLGP file:





Vocabulary conversion

If the "support completion allowed" check-box button is checked (for new project or, optionally for an existing project) then CoGui will convert a part of the Datalog imports into the vocabulary definition.

- Unary predicates are converted into concept types.
- Predicates with an arity > 1 are converted into relation types.
- Some rules are interpreted to order the concept types hierarchy and the relations hierarchies.
- Optionally, some rules are interpreted to define the relation types signatures.
- Optionally, some negative constraints are interpreted to define banned types.

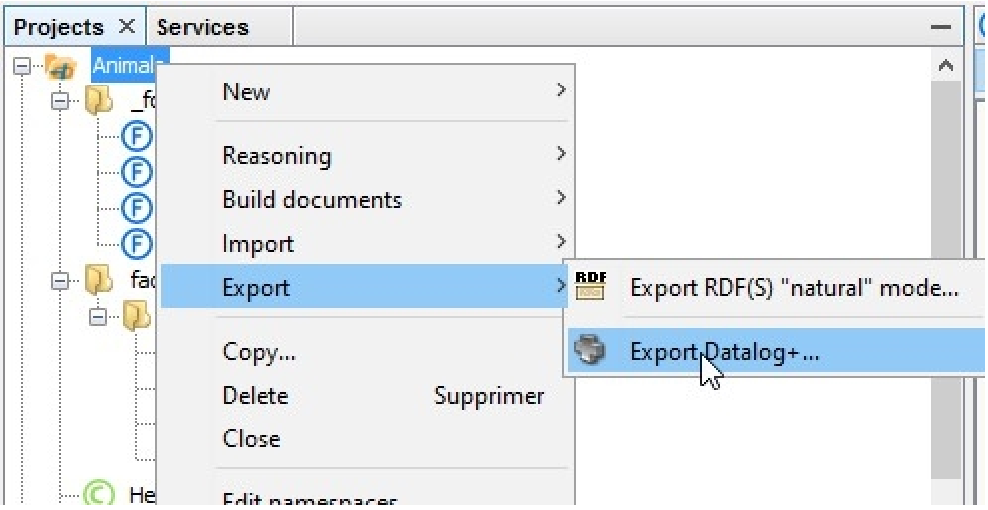
Note that unary predicates are converted into concept types on the CoGui model.

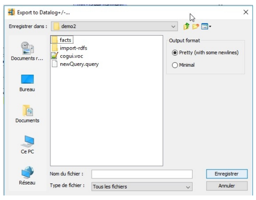
Some examples below:

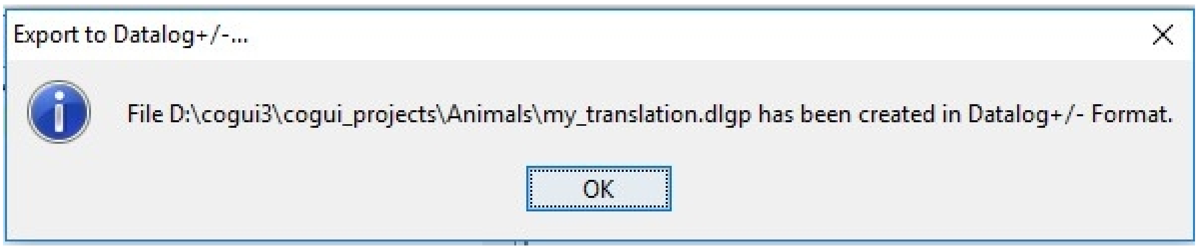
Datalog± expressions	Cogui conversion on the Vocabulary
<Human>(X):-<Man>(X).	From this rule, CoGui deduces a specialization relation between the concept types
!:-<Man>(X),<Woman>(X).	This negative constraint is transformed into a disjoint type.
parentOf(X,Y):-fatherOf(X,Y).	From this rule, CoGui deduces a specialization relation between the relation types.
<Man>(X):-fatherOf(X,Y).	The signature of fatherOf is modified by this rule.

Export to Datalog±

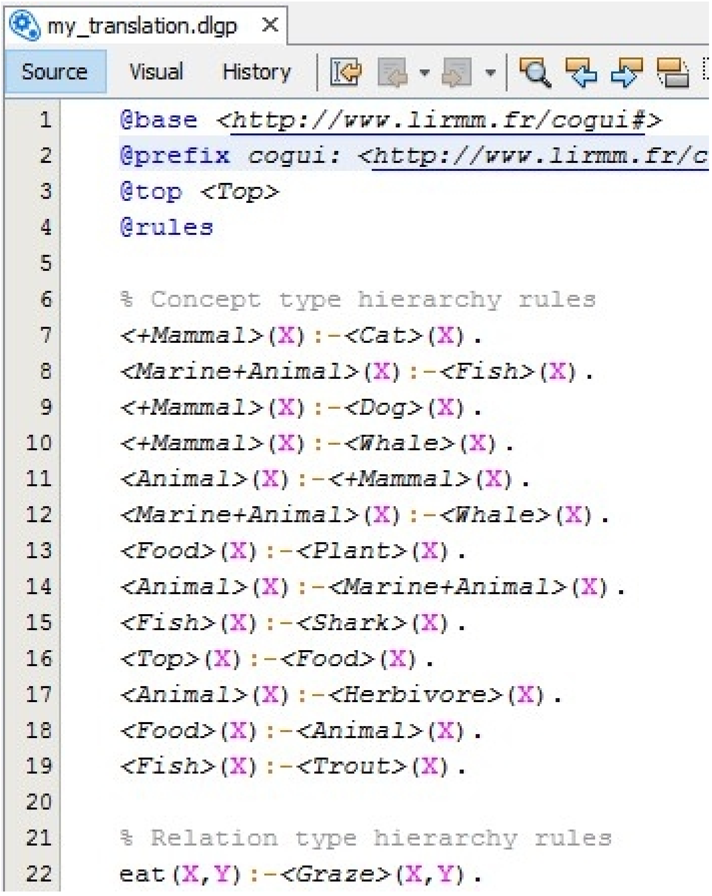
CoGui is able to export projects to Datalog± formats. All knowledge elements are converted (except scripts). Concept types are transformed into unary predicate. Relation signatures are converted into equivalent rules. Disjoint types are converted into negative constraints.
Individual labels are translated into Datalog constant name.There is no multilingual mechanism in Datalog, so predicates are built with the labels of the currently selected language.







DLGP files can be edited directly with CoGui editor:

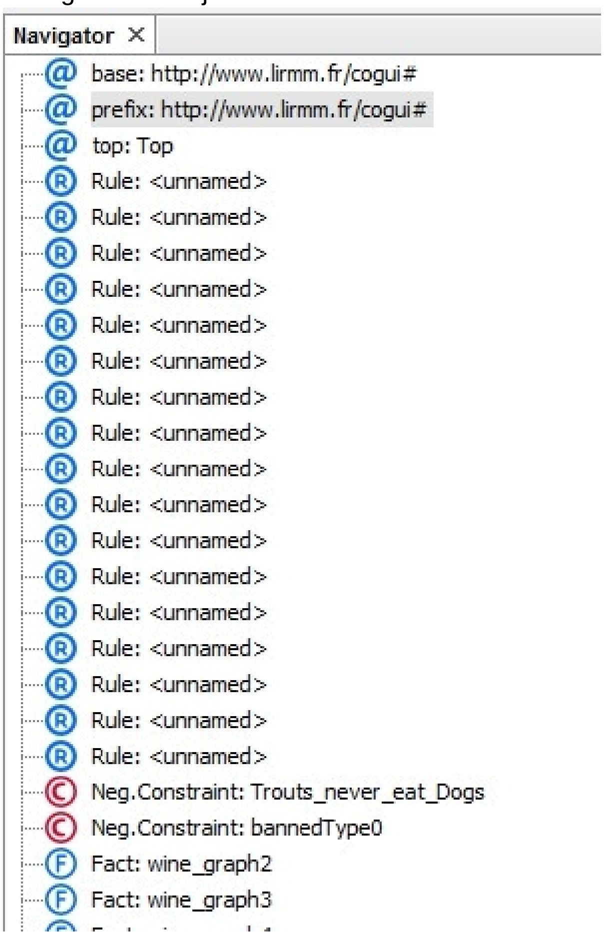



```
@base <http://www.lirmm.fr/cogui#>
@prefix cogui: <http://www.lirmm.fr/c
@top <Top>
@rules

% Concept type hierarchy rules
<+Mammal>(X) :-<Cat>(X).
<Marine+Animal>(X) :-<Fish>(X).
<+Mammal>(X) :-<Dog>(X).
<+Mammal>(X) :-<Whale>(X).
<Animal>(X) :-<+Mammal>(X).
<Marine+Animal>(X) :-<Whale>(X).
<Food>(X) :-<Plant>(X).
<Animal>(X) :-<Marine+Animal>(X).
<Fish>(X) :-<Shark>(X).
<Top>(X) :-<Food>(X).
<Animal>(X) :-<Herbivore>(X).
<Food>(X) :-<Animal>(X).
<Fish>(X) :-<Trout>(X).

% Relation type hierarchy rules
eat(X,Y) :-<Graze>(X,Y).
```

A navigator panel to browse through DLGP objects:



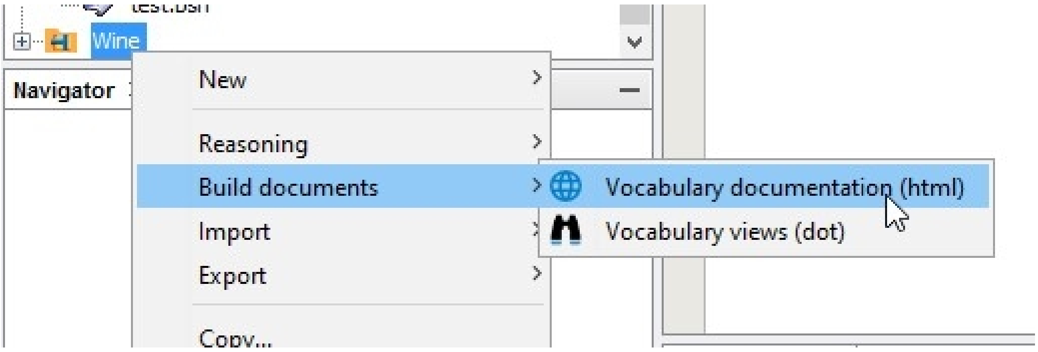
Building documents

For now CoGui offers two kind of tools to build documents.

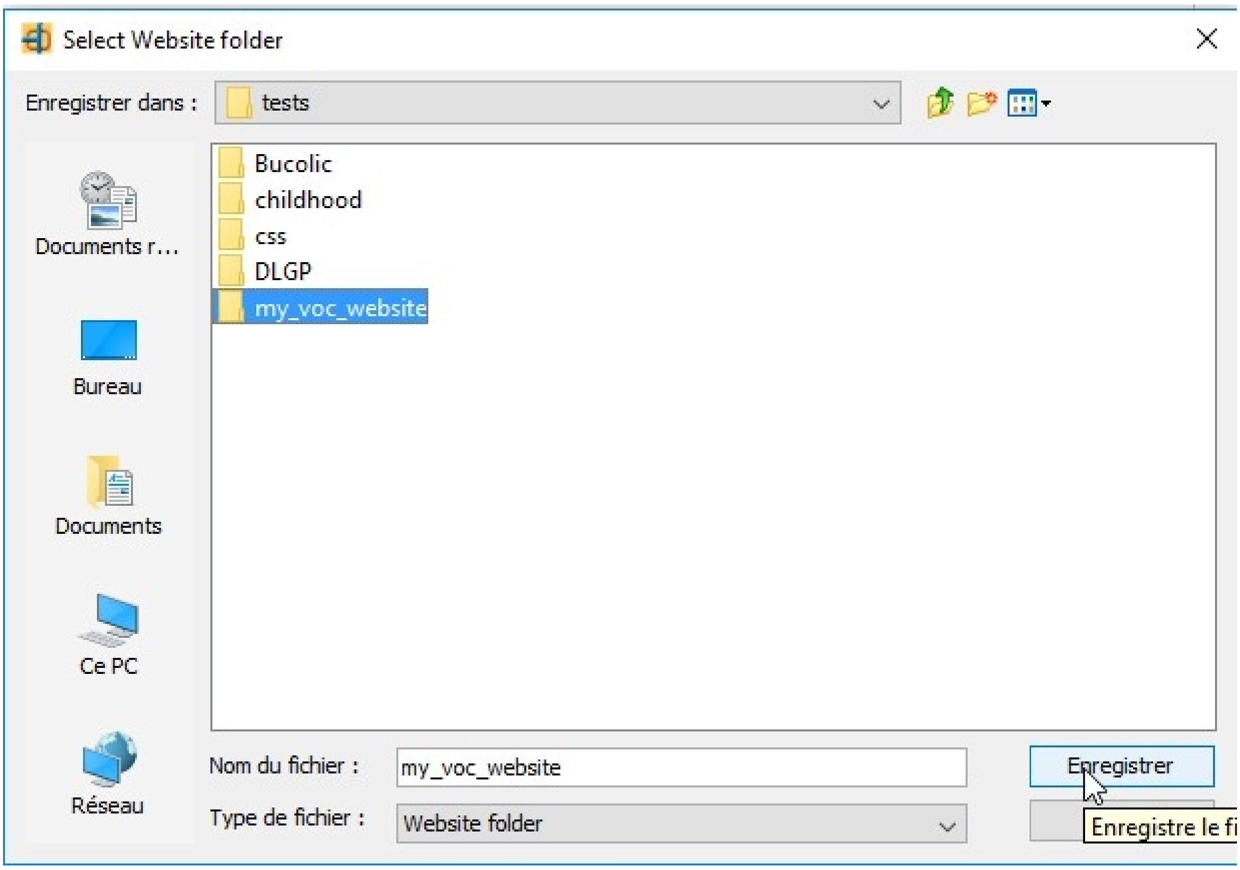
- Section Build vocabulary documentation explains how to build an HTML static website to document your CoGui project.
- Section Build vocabulary views presents a powerful tool for graphical representation of vocabulary using GraphViz software

Build vocabulary documentation

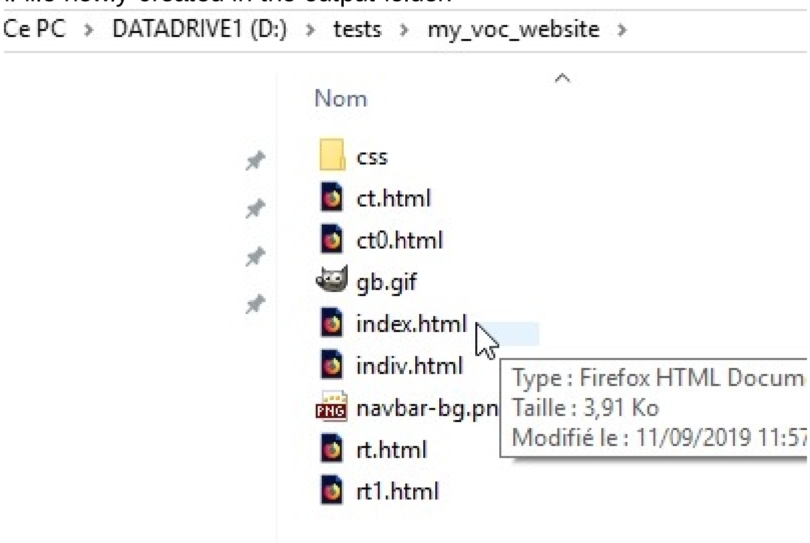
Select appropriate action in the menu of the project:



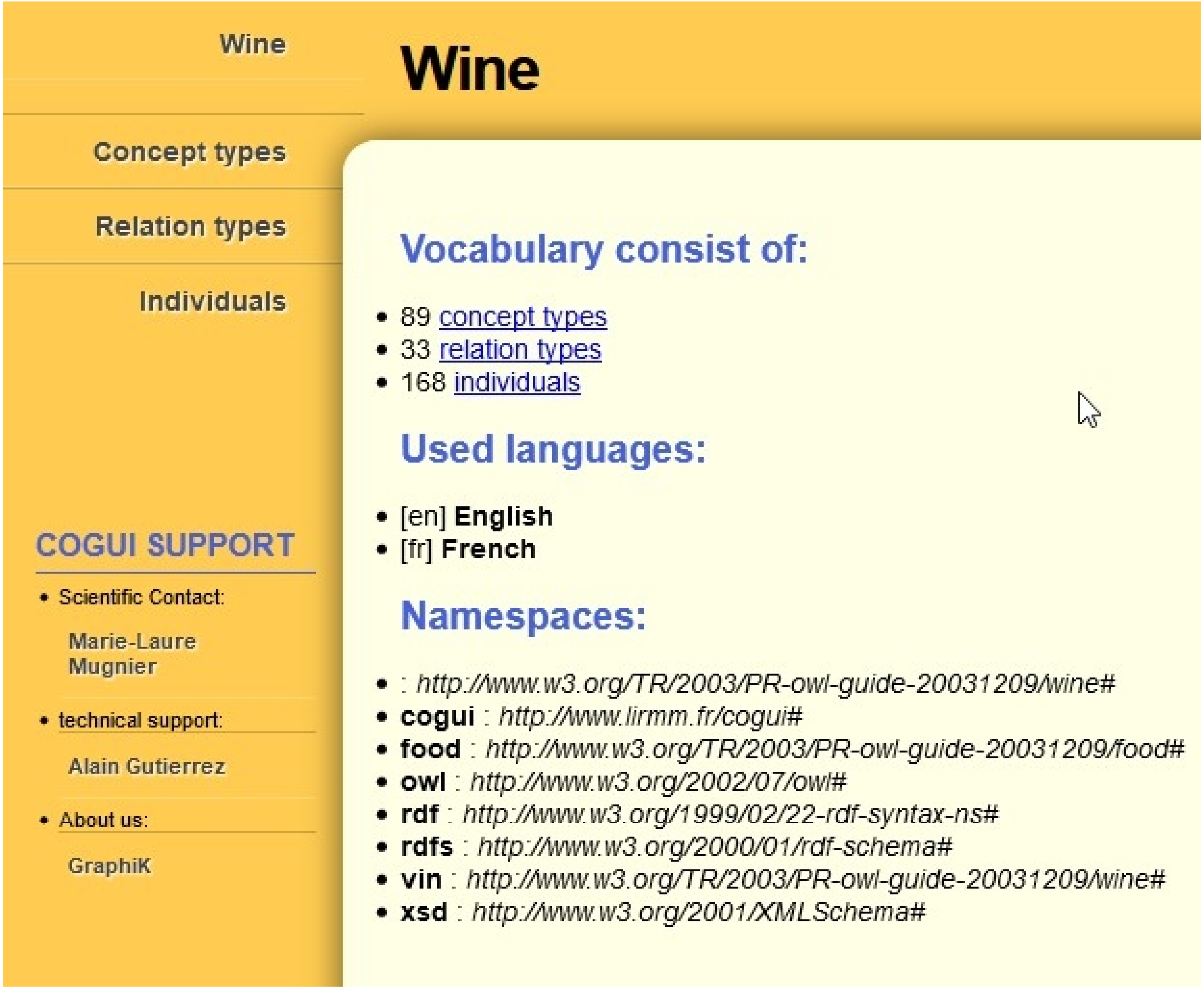
and Select the output folder :



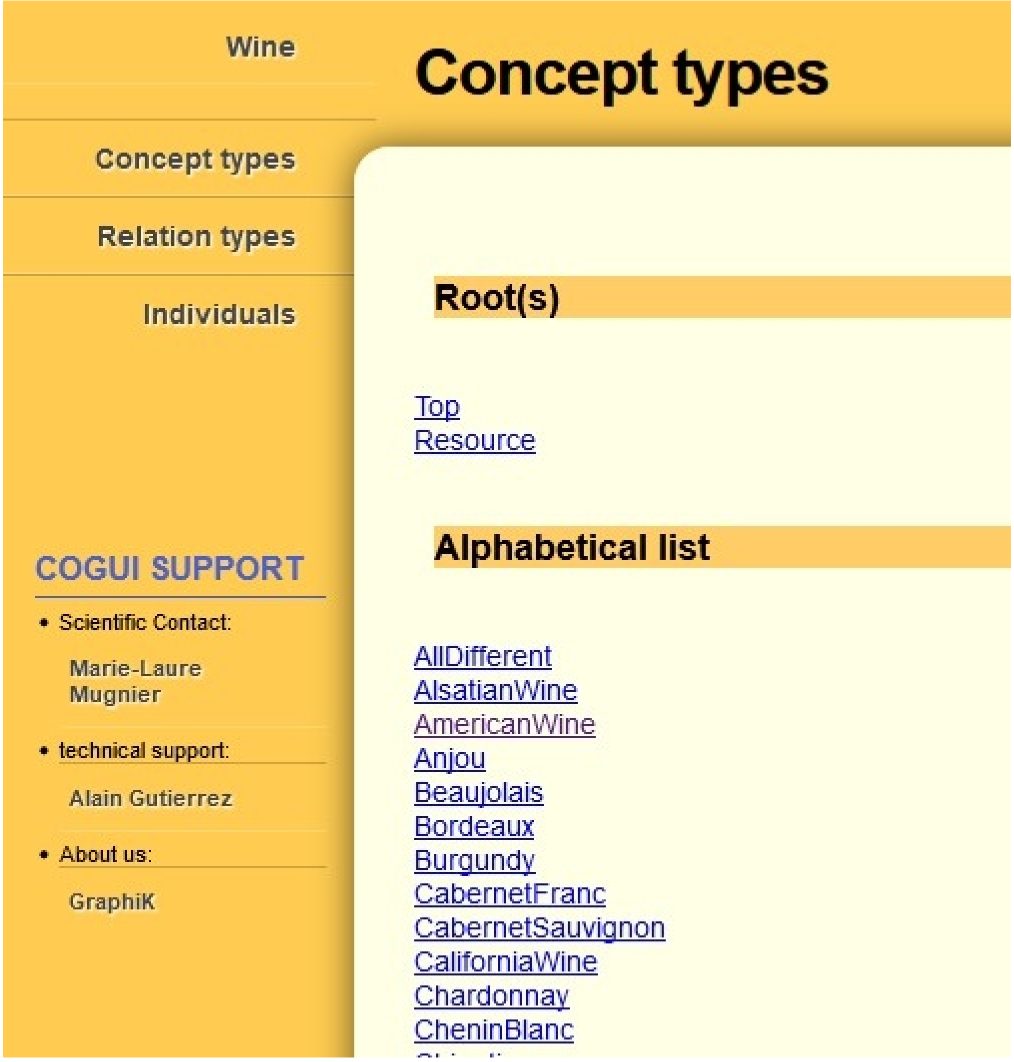
Open the index.html file newly created in the output folder:



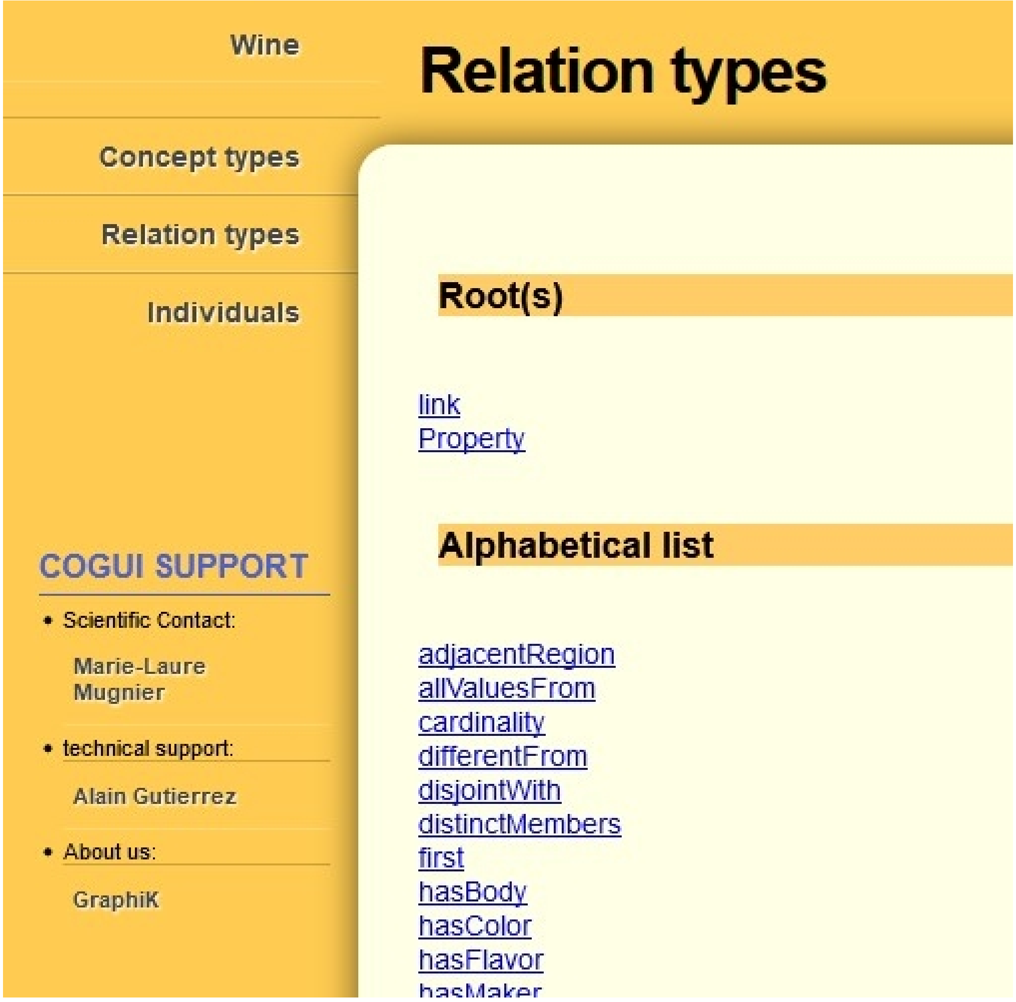
index.html page:



concept type list of the vocabulary:

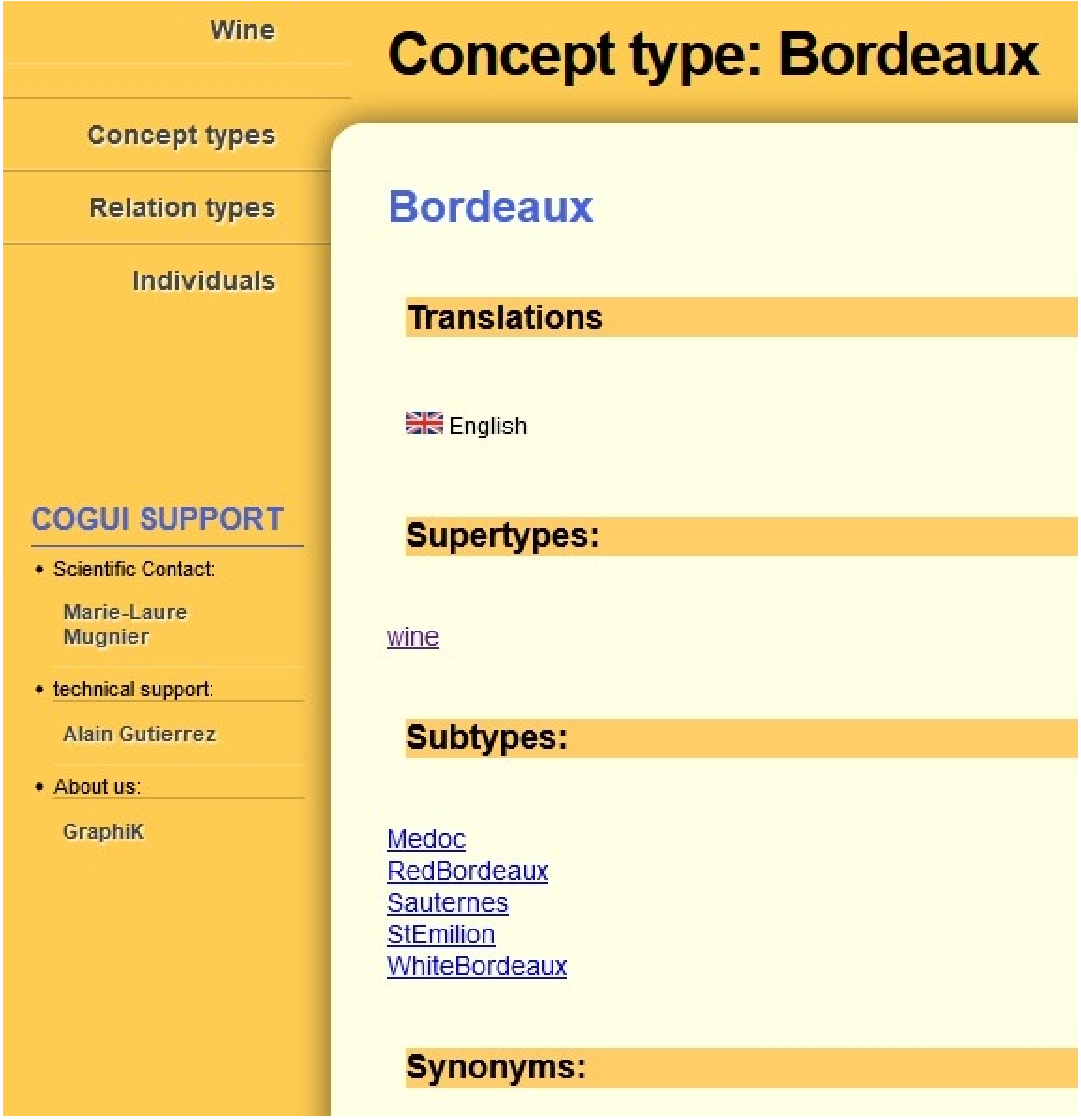


relation type list of the vocabulary:



a page for each relation type:

a page for each concept type:



Build vocabulary views

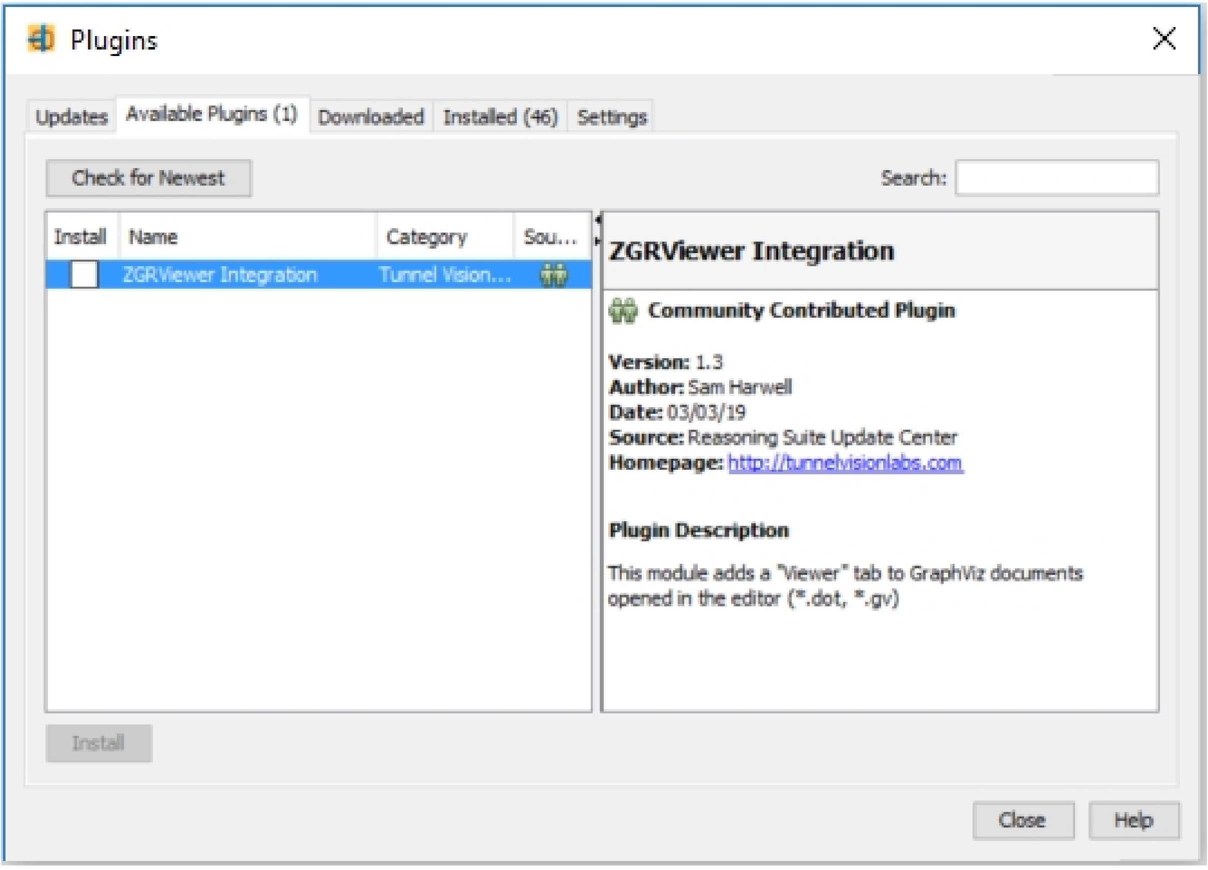
Pre-requirements

The Vocabulary Views produces formatted .DOT files
https://en.wikipedia.org/wiki/DOT_(graph_description_language)
DOT files are treated by the software GraphViz. The wizard doesn't need GraphViz to be installed to build DOT files. But installing this software on your machine will allow to edit and visualize these files directly inside CoGui. There are two steps to complete this installation:

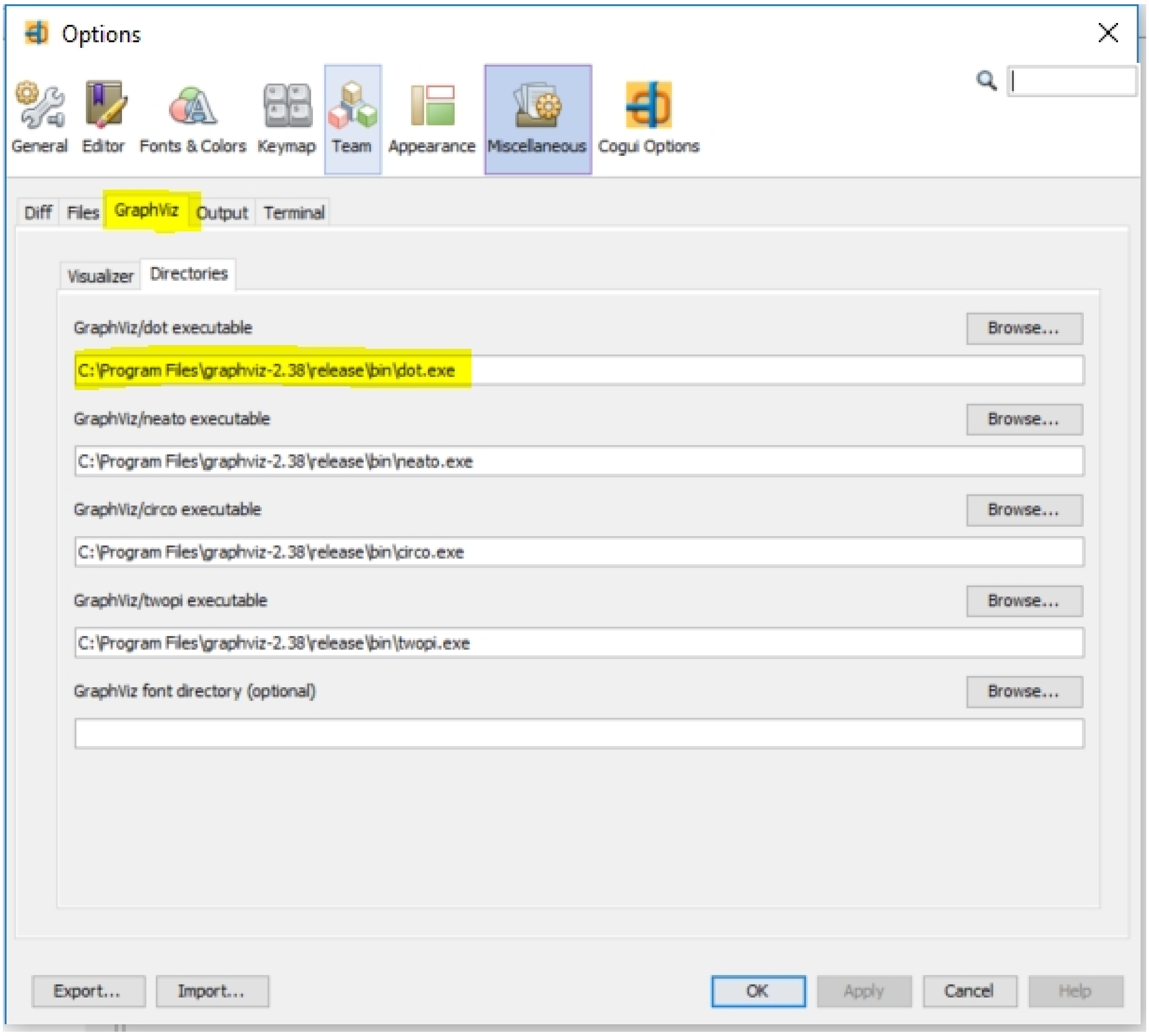
1) Install GraphViz from https://www.graphviz.org/download Note the location where the software is installed, you will need it later.
2) Install Plugin ZGRViewer.

The ZGRViewer plugin allows to edit the files (.DOT) produced by the wizard. To install it open the plugin manager of CoGui (menu Tools/Plugins).

The plugin "ZGRViewer Integration" can be found on the tab pane "Available Plugins". Install it:

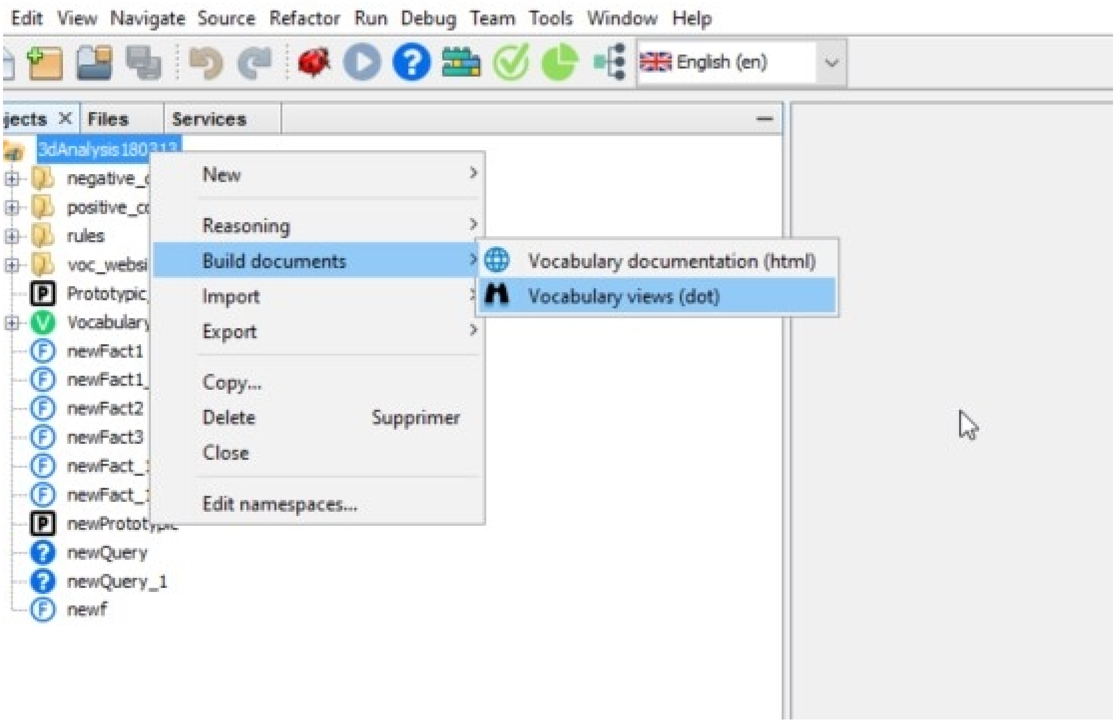


After relaunching CoGui, it remains to configure the plugin to use GraphViz previously installed. Access the options (Tools / Options / Miscellaneous / GraphViz / Directories menu) and specify the location of the executable files, especially the highlighted one in yellow, it is the DOT program (.exe under windows) which is used by the wizard .

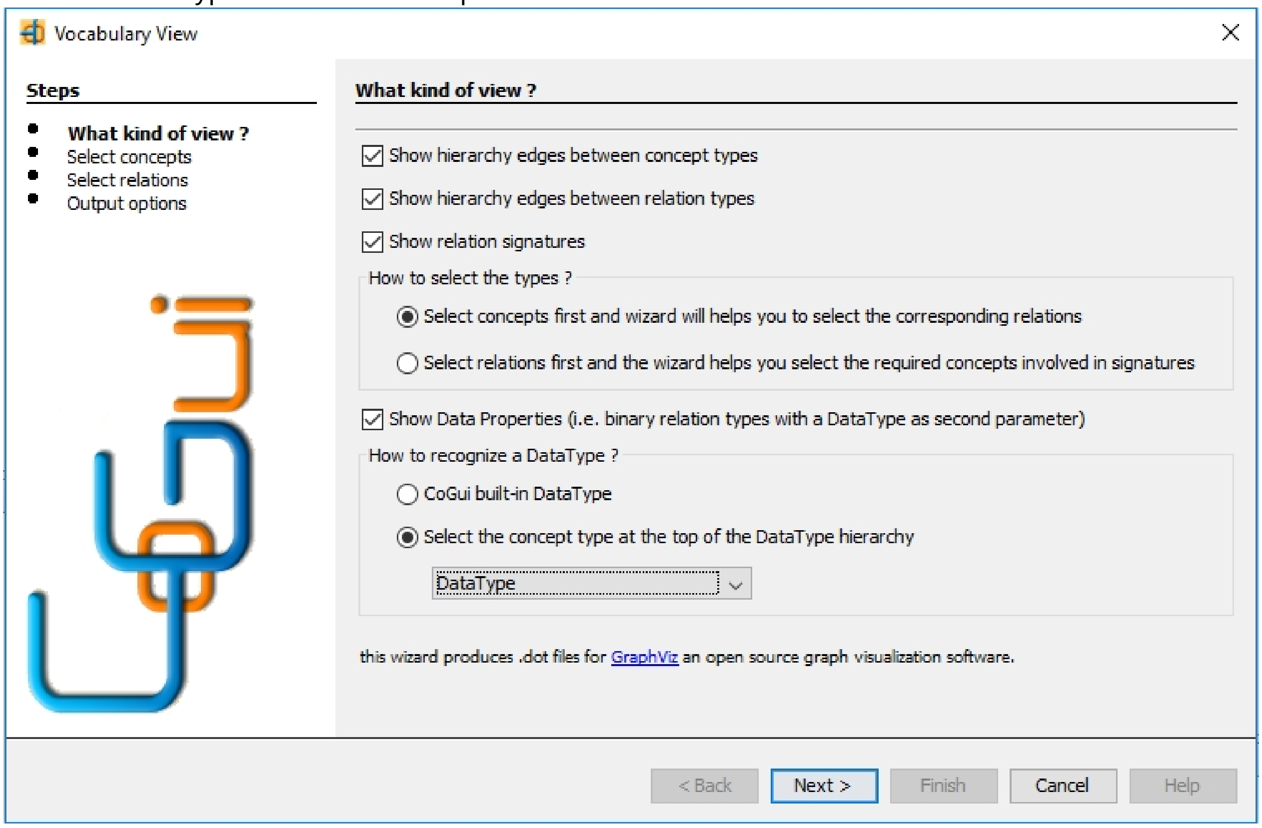


How to build Graphics

Select the project in the project tree and launch the command from the pop-up menu (fig.ci below) or from the Tools / BuildDocuments / menu



Definition of the type of document to be produced:



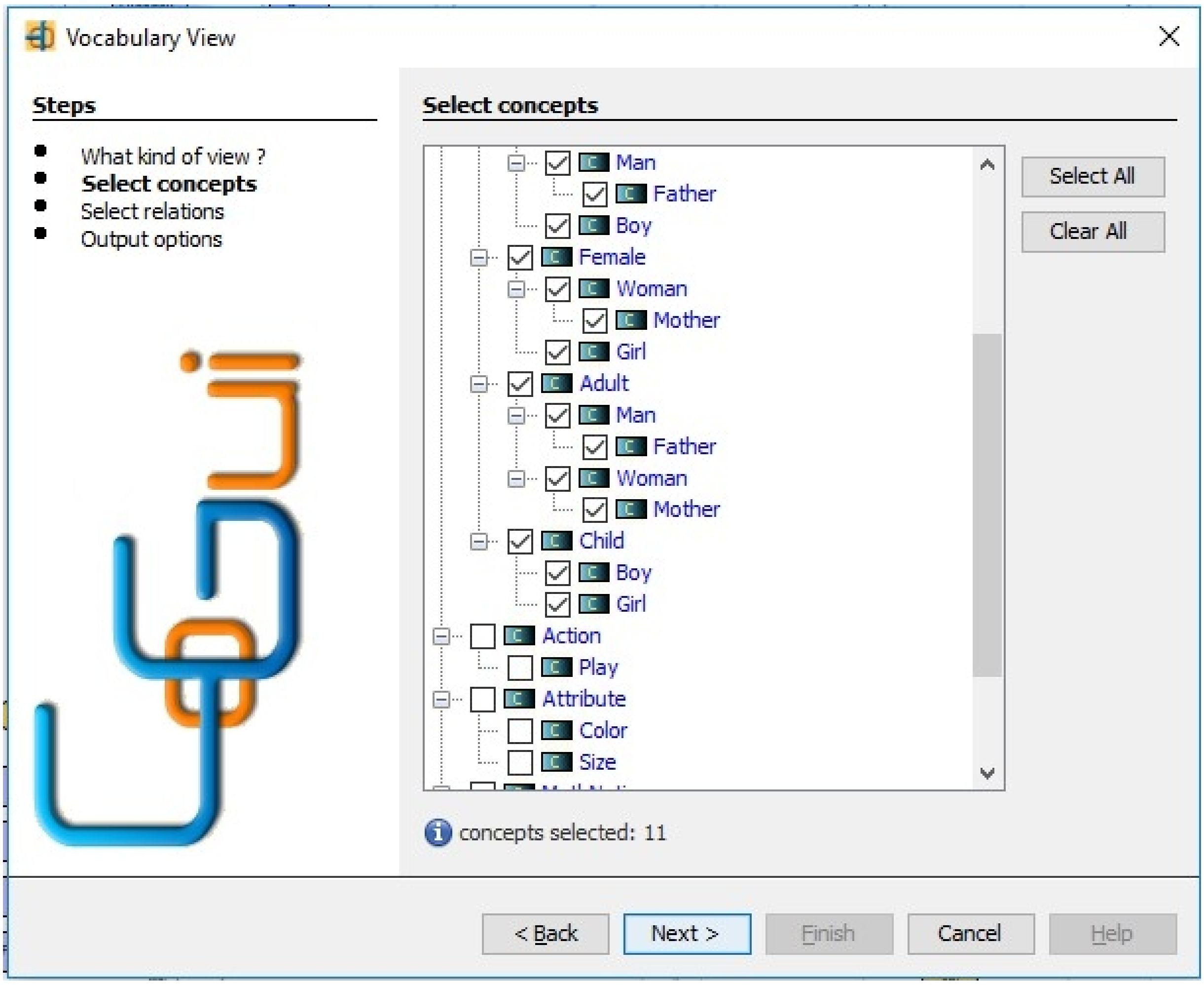
4 types of data are viewable and can be selected simultaneously:

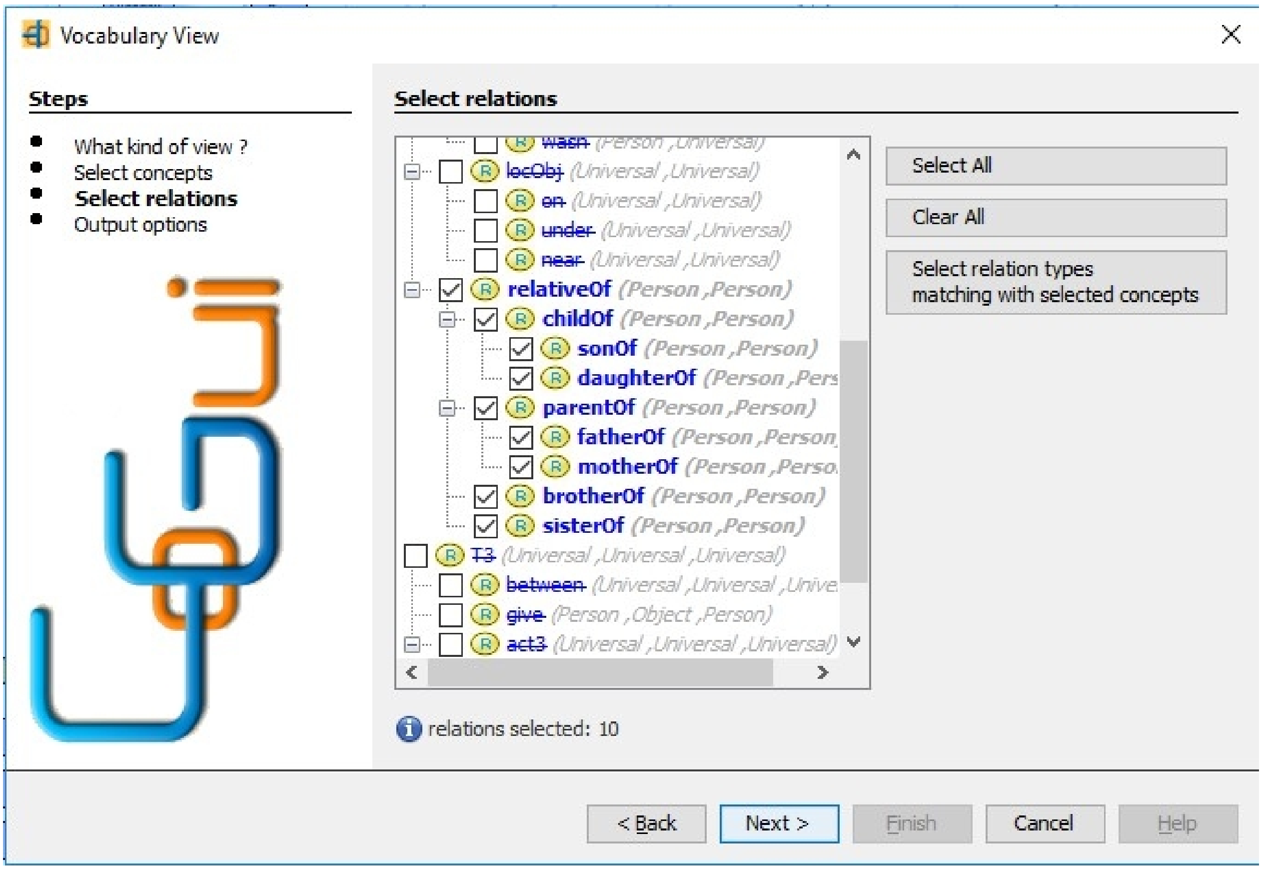
1. Materialization of "isKindOf" links between the types of concepts present in the document
2. Same for the links between the types of relations
3. The display of signatures that link relationship types to the types of concepts. This content introduces constraints in the selection of types. Indeed, it is necessary that the types of concepts present in the signature of a relation are also present in the document. To help the user, there are 2 selection scenarios:
   - Either one selects the concepts and the assistant will offer the possibility of automatically

selecting the types of relations whose signatures are compatible with the selected concepts

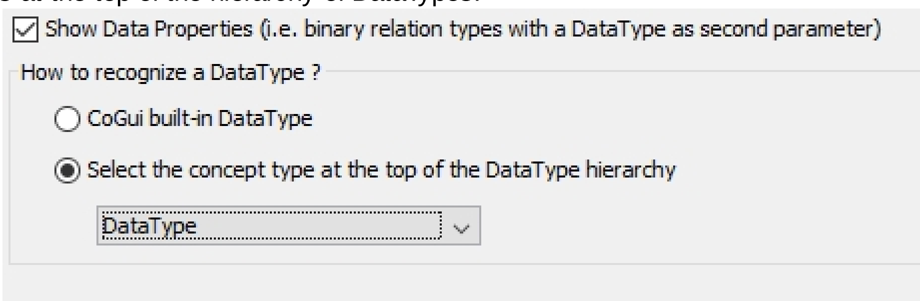
- Either one selects the relations and the assistant will offer the possibility of automatically selecting the types of concepts included in the signatures of the selected relationships

Note that we can go back (<Back) and combine the two scenarios to make his selection. In any case, the wizard will only display the final page if the constraints are respected (see figure below).

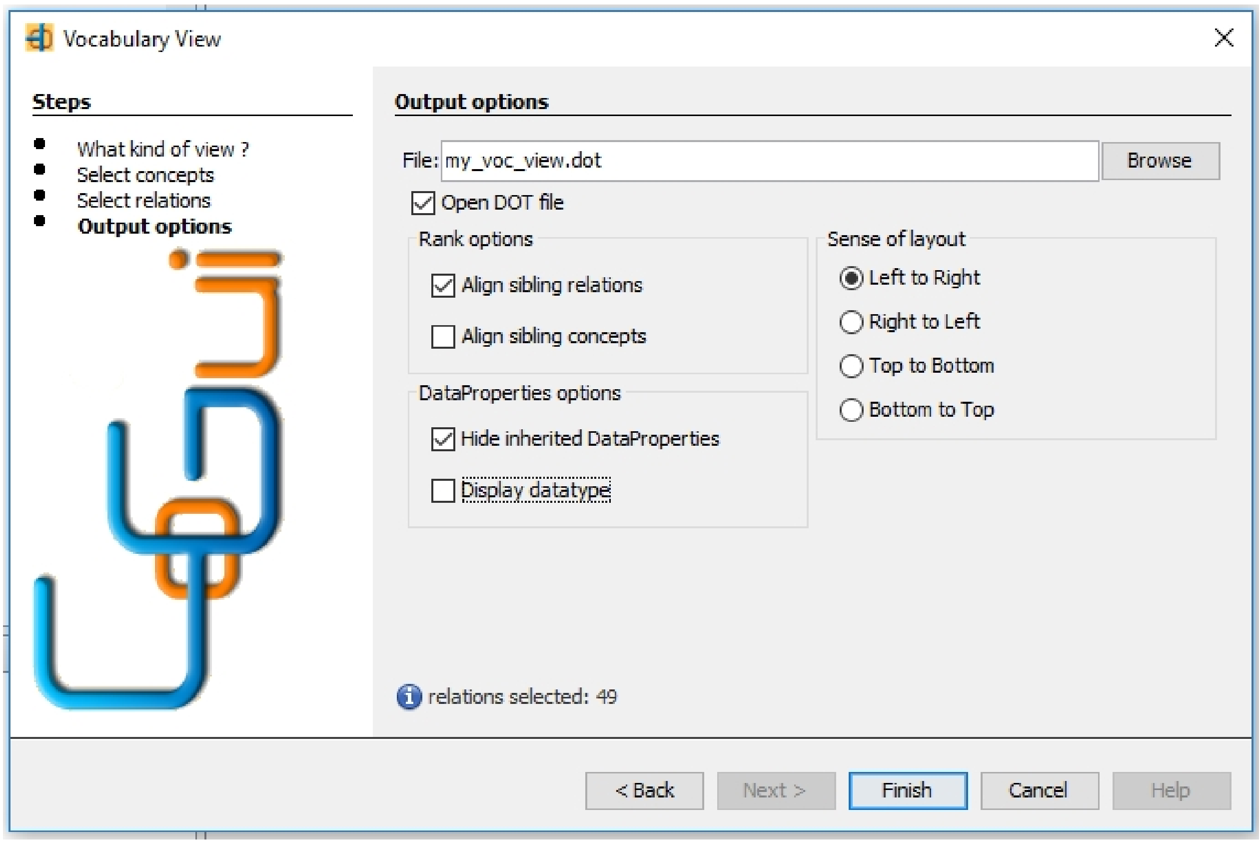




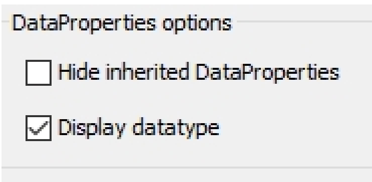
Display Data Properties:
The notion of Data Properties does not exist explicitly in CoGui but it can be assimilated to the types of binary relations whose second argument is a "DataType". There is an uncompleted mechanism in CoGui to manage the DataTypes (string, integer, float, boolean) but we will prefer, for the moment, to create its own DataTypes defined in the hierarchy of the concepts. This is the reason why, it is asked to specify the type of concept that is at the top of the hierarchy of DataTypes:

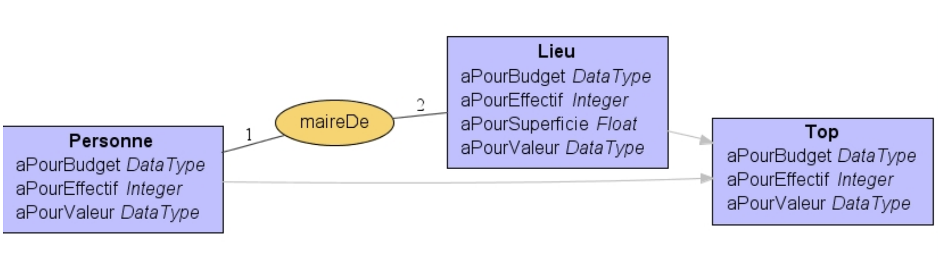


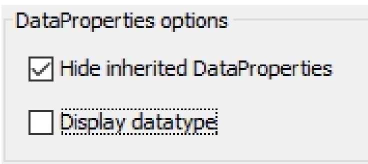
After choosing the type of document and selecting the types to display, you can complete the last part of the wizard that presents the output options, file name to produce and display options.

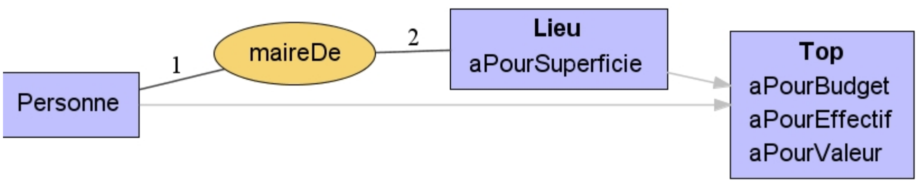


The "Rank" options force the page layout algorithm to align the types that have the same parent in the ontology.
The "DataProperties" options concern the display of this kind of relation inside the Concept vertex. The option "Hide inherited DataProperties" can make the schema lighter: DataProperties that appear in a concept type are not repeated in its sub-types.



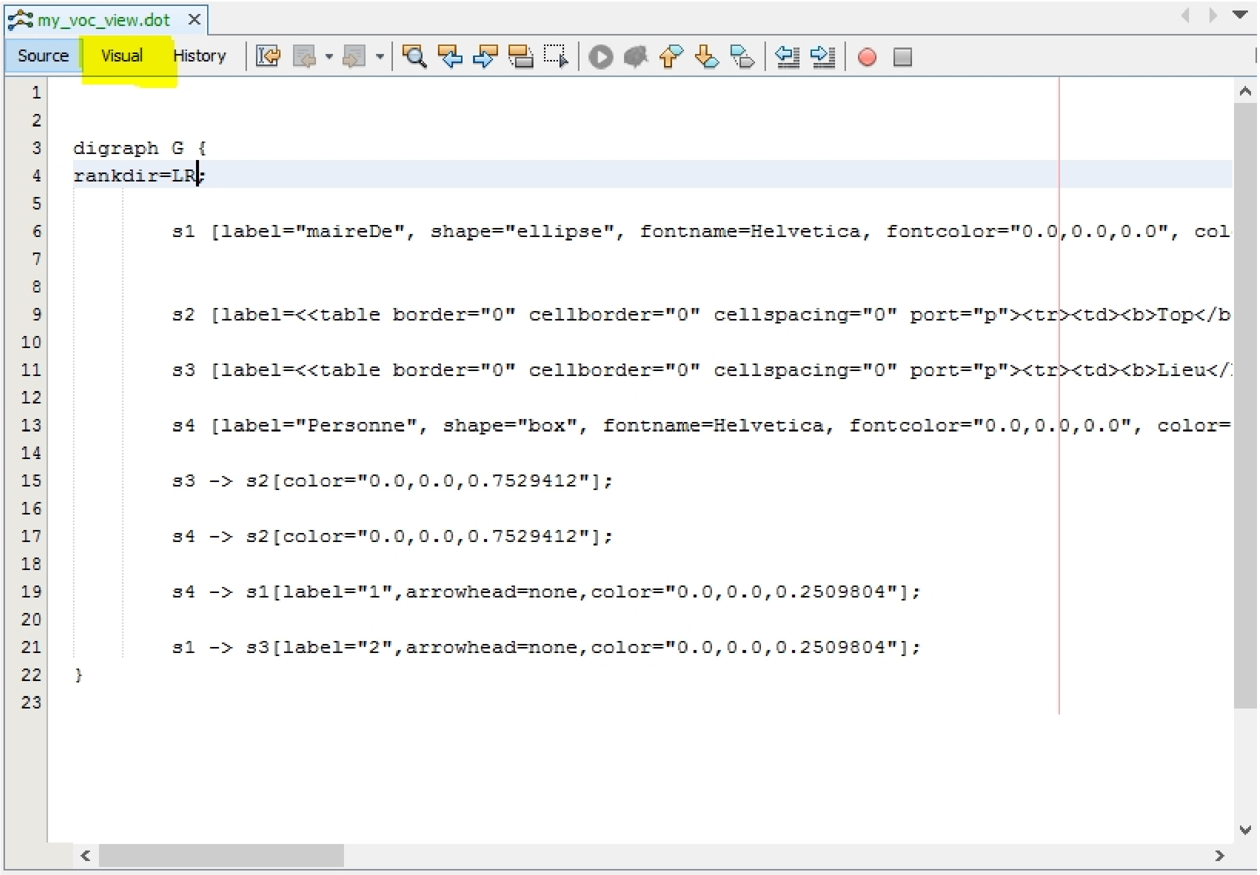




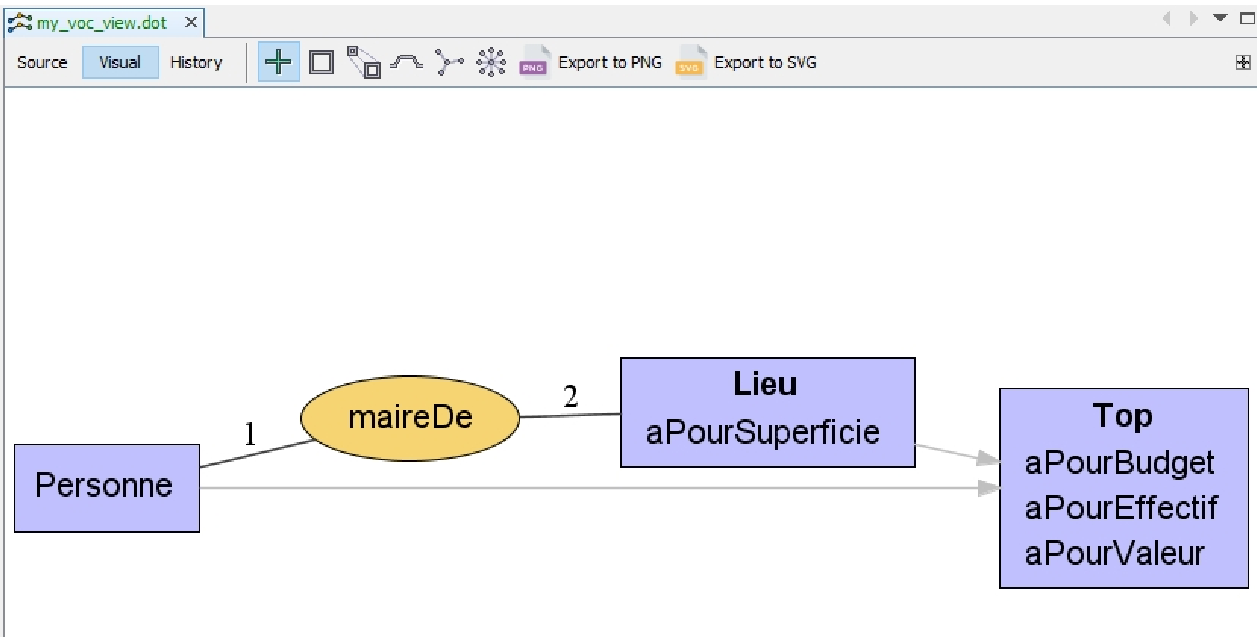


Finally, the "Sense of layout" options make it possible to influence the orientation of the document (Only influence because it must combine several types of edges and relations of opposite directions).

After having activated the "Finish" button of the wizard, it builds the diagram and opens the file in a window:



If the plugin has been correctly installed, the editor has an additional "Visual" pane highlighted in yellow. Operate the pane to access the graphical representation:



The toolbar offers several navigation and zoom modes. Two export commands are available for conventional images (.PNG) or vectorized drawings (.SVG) this last format is ideal to display on a web page or to make changes with software such as InkScape

Extending CoGui

CoGui propose 2 ways to extend its functionalities.

- Extending CoGui with Scripts
- Extending CoGui with Plugins

Scripts

The scripting language BeanShell (see http://beanshell.org) is embedded into Cogui application. This language was originally introduced because it is a good way to change the default behavior of the rules (see Scripted rules below). We finally decided to allow open use of this interpreter by introducing a new kind of objects in the Cogui projects : executable scripts.
Scripts greatly enhance the user capabilities. For example, by providing a way to chain graph operations proposed by Cogui or include some new graph algorithms. More generally scripts can compensate the lack of a plugin architecture. Cogui Java classes and objects are exposed to the script language so that user can access on public members of existing objects and also instantiate classes to introduce new objects.

Basics

BeanShell can dynamically execute full Java syntax, as well as loosely typed Java and additional scripting conveniences. Documentation about BeanShell can be found here. Two kinds of commands are available in cogui scripts.

- Some native BeanShell commands are described further. All BeanShell commands are documented: BeanShell commands.
- Cogui propose its own commands. See Cogui commands further.

Before script execution, 3 global variables are instantiated. These variables give access to the current objects loaded in Cogui.

Global variables

Name	Description
`_PRJ`	represents the current project (an instance of `fr.lirmm.graphik.cogui.edit.p`
`_KB`	equivalent to `_PRJ.getKnowledgeBase()` return, it represents the current Knowle
`_VOC`	equivalent to `_PRJ.getVocabulary()` return, it represents the current vocabulary (

First use, the most natural, is to use scripts to access project existing objects in order to read, analyze or modify them. Global variables are pointing to objects containing all methods necessary to obtain such access. Once the concerned object is obtained, we can use the public methods as described in the API cogui.core.model. Suppose, for example that we want to visit all the vertices of a graph and count the individuals that it references.

```
nbGeneric=0;
g=getFact("set1/fact_1");
it=g.iteratorConcept();
while(it.hasNext())
        if(it.next().isGeneric())
               nbGeneric++;
print("graph "+g.getName()+" contains "+nbGeneric+" generic
concept(s)");
```

Access to vocabulary elements

Element	How to obtain
The concept type hierarchy	`_VOC.getConceptTypeHierarchy()` or `_VOC.getHierarchy(Vocabulary.CONCEPT_TYPE)`
The relation type hierarchy	`_VOC.getRelationTypeHierarchy()` or `_VOC.getHierarchy(Vocabulary.RELATION_TYPE)`
The individuals	
	`_VOC.getIndividuals()`

Work with CoGui core package

Exploring graphs

CoGui propose several assistants for querying, applying rules, checking and analyzing graphs. All these operations are based on an homomorphism search provided by the CoGui solver (Solver5).
CoGui provides a user friendly search command to explore graphs programmatically.
As an example, we will write a script which produce the list of the parents in a person relationship knowledge base. At first, user write a corresponding query.

```
listener()
{
      projectionFound(engine,proj)
         {
          print("coucou="+toto);
         } return this;
 };
myListener=listener();
search( getFact("set1/fact_1"), getQuery("set1/query_1"),
myListener);
```

Work with the vocabulary

Vocabulary is a composite class containing primarily a Hierarchy and some Translator(s)
Code below demonstrates how to define types in current vocabulary

```
/* create a concept type */
_VOC.addConceptType("ct_1", "top", "the root type", "en");
/* create a binary relation type with the signature (top,top)
_VOC.addRelationType("rt_1","link","the root type for binary
relation types","en");
_VOC.setSignature("rt_1",new String[]{"ct_1","ct_1"});
```

```
/* create a nesting type */
_VOC.addNestingType("nt_1","nesting","the root nesting
type","en");
```

Translator

Translator class is a mapping between the type identifier and a pair (label,language) .
Both labels and descriptions of each kind of types (concept, relation and nesting) are represented by instances of Translator class.
Translator can be defined to guarantee the uniqueness of labels. Unlike description translators, label translators use this option. So, for the same language, two different identifiers cannot be associated to the same label.
With this property the method getId (String label , String language) can be used to search id efficiently on a double map.

```
/* code below is equivalent to this Vocabulary call:
** voc.addConceptType("ct_1", "top", "the root type", "en");
*/
Translator
labelTranslator=voc.getTranslator(Vocabulary.CONCEPT_TYPE);
Translator
descrTranslator=voc.getDescriptionTranslator(Vocabulary.CONCE
PT_TYPE);
labelTranslator.addLabel("ct_1", "top","en");
descrTranslator.addLabel("ct_1","the root type", "en");
/* id of types can be found with label */
System.out.println("id="+voc.getTranslator(Vocabulary.CONCEPT
_TYPE).getId("top","en")+
"
label="+voc.getTranslator(Vocabulary.CONCEPT_TYPE).getLabel("
ct_1","en"));
```

Use `getDefaultLabel(String id)` for monolingual vocabulary. Cogui also use monolingual Tranlator to store individuals.
Code below print labels of a type. Method `getLanguage(String id)` search every translations for a type.

```
for(String lang:labelTranslator.getLanguages("ct_1"))
System.out.println("label="+labelTranslator.getLabel("ct_1",l
ang)+" lang="+lang);
```

Hierarchy

When Translator(s) store labels and other informations about types, Hierarchy complete the model with a directed graph representation of the kindOf relationShip between types.
Vocabulary give accessors for each Hierarchy instance:

- `getConceptTypeHierarchy()` equivalent to `getHierarchy(Vocabulary.CONCEPT_TYPE)`
- `getRelationTypeHierarchy()` equivalent to `getHierarchy(Vocabulary.RELATION_TYPE)`

Example below show how to access the hierarchy and add two concept types ct_1 and ct_2 where ct_2 is a kind of ct_1:

```
Hierarchy ctH=voc.getConceptTypeHierarchy();
ctH.addVertex("ct_1");
ctH.addVertex("ct_2");
ctH.addEdge("ct_2","ct_1");
System.out.println("ct_2 is kind of ct_1 ?
"+ctH.isKindOf("ct_2","ct1"));
```

Hierarchy give efficient access to the graph to iterate vertices or incoming and outgoing edges.

- `edgeSet()` and `iteratorEdge()` to explore all edges
- `edgeSet(String)` and `iteratorEdge(String)` to access edges of a given vertex
- `incomingEdgeSet(String)` and `iteratorIncomingEdge(String)` to access incoming edges of a given vertex
- `outgoingEdgeSet(String)` and `iteratorOutgoingEdge(String)` to access outgoing edges of a given vertex
- `vertexSet()` and `iteratorVertex()` to explore all vertices

This is useful to write your own algorithm. But some graph algorithm are already implemented by Hierarchy class
most of them are wrapped from JGraphT library Hierarchy can be overloaded to access other algorithm not already used by Cogui.
Following tools are proposed by Hierarchy class:

- about transitivity: a closure and a transitive reduction
- a cycle detector and a method to iterate all connected components
- several methods to compare (`isKindOf`) and normalize sets of types (`normalize(String[])` and `isRedundant(String[])`)

Build a fact graph

After defining a Vocabulary instance, we will programmatically build facts based on this vocabulary. For this purpose the KnowledgeBase class was designed to associate a vocabulary with a set of facts and possibly with rules, constraints, prototypics etc.

KnowledgeBase

KnowledgeBase is a composite class designed to associate a vocabulary with:
- facts graphs
- queries
- rules
- positive constraints
- negative constraints
-

A Knowledge instance is used to store and give access to the graphs directly or throw access to GraphSet instances

```
KnowledgeBase kb=new KnowledgeBase(voc);
/* this two instructions below are equivalent */
kb.getFactGraph("g_1");
```

Now we will create and populate a fact graph to store it inside the knowledge base.

CGraph

CGraph is composed by a set of Concept and a set of Relation mapped with their keys and a Multigraph instance represent the graph itself. Code below shows how to create and populate a CGraph instance.

```
/* create the graph */
CGraph graph=new CGraph("g_1", "my graph name", "my_facts",
"fact");
/* create the concepts and a relation */
Concept c1=new Concept("c_1");
Concept c2=new Concept("c_2");
c1.setType("ct_1");
c2.setType("ct_2");
Relation r1=new Relation("r_1");
r1.setType("rt_1");
/* populate the graph with vertices */
graph.addVertex(c1);
graph.addVertex(c2);
graph.addVertex(r1);
/* add edges */
graph.addEdge(c1.getId(),r1.getId(),1);
graph.addEdge(c2.getId(),r1.getId(),2);
/* add the fact graph to the knowledge base */
kb.addGraph(graph);
```

CoGui commands

Name	Description

`applyRule(CGraph graph,Rule rule)`
`applyRule(CGraph graph,Rule rule,int limit)`
`applyRules(CGraph graph,ArrayList rules,int limit)`
`applyRules(CGraph graph,ArrayList rules,int limit,boolean scripted)`
	Apply the rule(s) on the graph until limit level and until saturation if limit==-1 (default).
	Note: the graph is directly modified by the action of rules. Use cloning to preserve the original graph with CGraph.clone() function.
	If scripted is set to true then the scripts contained in scripted rules are called. See scripted rules for details.

`CGraph getFact(String name)` `CGraph getFact(String name,String set)`	Return corresponding fact graph. Equivalent to _KB.getFactGraphSet().getByLabel(name,set) Example: getFact("set1/fact_1") or getFact("fact_1","set1").
`CGraph getPConstraint(String name)` `CGraph getPConstraint(String name,String set)`	Return corresponding positive constraint. Equivalent to _KB.getPConstraintGraphSet().getByLabel(name,set) Example: getPConstraint("set1/pconstraint_1") or getPConstraint("pconstraint_1","set1").
`CGraph getPrototypic(String name)` `CGraph getPrototypic(String name,String set)`	Return corresponding prototypic graph. Equivalent to _KB.getPrototypicGraphSet().getByLabel(name,set

	) Example: getPrototypic("set1/proto_1") or getPrototypic("proto_1","set1").
`CGraph getQuery(String name)` `CGraph getQuery(String name,String set)`	Return corresponding query graph. Equivalent to _KB.getQueryGraphSet().getByLabel(name,set) Example: getQuery("set1/query_1") or getQuery("query_1","set1").
`search(CGraph graph, CGraph query, bsh.This pListener)`	pListener is scripted object containing a callback method: projectionFound(Solver solver,Projection proj).

Plugins